On computing discrete logarithms: large prime(s) variants

Holt, Andrew James

Award date:
2003

Awarding institution:
University of Bath

On Computing Discrete Logarithms: Large Prime(s) Variants

submitted by

Andrew James Holt

for the degree of PhD

of the

University of Bath

2003

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Summary

This thesis investigates the advantages to be gained from the application of large prime variant techniques to the index calculus method for computation of discrete logarithms modulo a prime p. Such techniques have been applied with great success to the index calculus method as applied to the integer factorisation problem, but are rarely mentioned for the analogous discrete logarithm application.

The thesis follows various implementations through from parameter choice to final discrete logarithm computation. We firstly show how one may make small but practical savings in the linear algebra step of the index calculus method by suitable choice of the generator of the nonzero elements of the finite field under consideration. We then move on to examine the practicalities of applying large prime variant techniques at each stage of the method.

The main focus of the work is to highlight the differences between the well documented application of large prime variant techniques in factoring, compared with their use in the discrete logarithm case. We demonstrate how the standard graph theoretic methods of Lenstra and Manasse [75] may be adapted to the discrete logarithm situation by considering the nature of cycles, and show how one may resolve such cycles without the need to solve a linear system, as has been the case in the few implementations discussed in the literature. We also illustrate certain situations that can occur which do not allow us to achieve the same yield as in factoring applications. We then consider the impact of using more than two large primes, and show how the factoring methods of Leyland et al. [81] adapt to the discrete logarithm case, such that we can resolve all but a fraction of cycles found.

We subsequently discuss how the divergence of the index calculus method for factoring and for discrete logarithm computation requires us to consider more than simply minimising solution time in the linear algebra step – we must also try to maximise the number of values we can recover in our solution vector. We show how large prime variants provide both benefits and drawbacks in this context, and demonstrate how their use can speed up final discrete logarithm computation quite considerably.

Acknowledgements

This page started as a basic list of names. Then I thought, hey, it's my thesis, what the hell. I have had a great time here and it would not have been possible without the help of a lot of people who have had a major influence on this work and on my time in Bath generally. Three years is a long time, especially when you are staring at several thousand fairly large numbers day in, day out...to be perfectly honest my current attitude is 'thank God for that', and indeed that I have come out the other side in much the same shape in which I went in. Well, more or less.

Top of the tree on the work side of things is my supervisor James Davenport. Quite how James manages to have time for research whilst appearing to be responsible for the running of practically everything at Bath I have no idea. It has been a privilege to work with James for three years, and I am very grateful for his encyclopedic knowledge, expert help and quite remarkable patience, not to mention his offer of a position in the first place. Cheers James. Thanks are also due to the maths/CS department(s) for giving me a scholarship – sorry for spending most of it on a bike.

Over the course of three years one bumps into a variety of people, and I would like to thank all who have commented on my work or helped me in any way, however small – it meant a lot and gave me confidence when I needed it. So thanks to Emma Jones, Russell Bradford, John ffitch, Ceri Fiddes, Ben Mankin, JF Williams, Brian LaMacchia, Peter Montgomery, assorted anonymous referees, and all at RWCA '02 and SECANTS 18; especially Nigel Smart, Steven Galbraith and John Pollard for kind feedback on my talks there. Special thanks to Damian Weber and Paul Leyland for providing datasets used in chapter 3 and for advice on the work contained in chapter 5 of this thesis. Last but not least, thanks to the maths/CS department(s) both here and at St. Andrews and Bristol; especially Cath Gerrard and the rest of the Solar Theory group at St. Andrews, and to Ivan Graham and Edward Fraenkel at Bath. I am also very grateful to the CS support staff for their help, and for not laughing at my stupid questions (or at least for waiting until I had left the room).

A different kind of work also deserves a mention – thanks to IBM Global Services, née PricewaterhouseCoopers, for allowing my career break. Many thanks to Andy Gibbs for arguing my case, and to Steve Kingston and John Blackburn for authorising it. I made some good friends at PwC and am sorry that not all – in fact, I think, hardly any – will be there when I go back. Thanks to all of the class of 23/11/98, especially Laurence for writing me a reference, and to my team on Xerox in 2000. Seems like anyone who got put in charge of me while I was away has been made redundant, so, err, yeah, sorry about that.

Thanks to my mates here in Bath, particularly my fellow PhD students for keeping me roughly as sane as I was when I arrived. Special thanks to Steve, Sarah, Marc, John, Adam and Natee for all those tea breaks. Apologies to the maths department football

team – especially Mark, Rich, Pete, Alex and John, my fellow (sometime) captains – and to the kickboxing club for my lack of talent. Thanks to Rachid and Eamonn for foolishly agreeing to go running/cycling with me, and demonstrating what one can achieve if one consumes enough painkillers and home made performance enhancing drugs. Sorry for saying that the Macc half marathon course was flat...you'll laugh about it one day. I am also very grateful to Robin Jackson for expert and entertaining piano tuition – apologies for being so ham-fisted.

There are of course a hell of a lot of other people due a mention. Special thanks to my friends who supported me in all this, especially Tom, Denise, Rick and Mark. Thanks to my flatmates – JF, Tadeja, David, Steve, Alex, Aliki and Amandyne for putting up with my domestic eccentricity. Thanks to Stan, Norma, Lindsey and Dave for being so kind and supportive. And nearly finally, thanks most of all to my family. Unshakable faith in one's abilities can sometimes seem kind of hard to live up to, but it is nice really, and I will forever be grateful to my mum and dad and brother for supporting me, for always believing in me and never once failing to be there for me when I needed it. Thanks to the rest of my family too, particularly Madge and Alan. Sadly some people I'd like to thank aren't here to see the end of this – quite possibly like some people who started reading it – but I guess they are really, and know I don't forget.

Last but most, I'd like to thank my Kirsty – without any doubt at all I could not have done any of this without you, and no, I don't care if this embarrasses you. I'm sorry for being away. OK, you'll be glad to know I'm done. Anyone I've forgotton?...probably. My apologies. Mum, Dad, Kirsty – you're the best; this is for you.

AJH – November 2003

Contents

List of Figures

List of Tables

List of Algorithms

Part I

Introduction and Background

Chapter 1

Introduction

This thesis is concerned with techniques for the computation of discrete logarithms modulo a prime. While this topic is of mathematical and computational interest in its own right, a large amount of research on this topic over the last 25 years or so has been motivated by its practical application in modern cryptography. In this chapter, in order to get an idea of the scope of the subject, we first give a brief overview of cryptography, before moving on to discuss the structure of the thesis.

1.1 Cryptographic background

1.1.1 A brief history

One can argue that there has been a need for secret communication ever since mankind learned to communicate. With the arrival of the written word, this need led to the development of formal methods for the disguising of the meaning of messages and communications. Both influential individuals and governments found the need to protect their own sensitive information. Techniques varied from actually disguising the fact that any information was present via *steganography* - using invisible ink, or hiding a message such as in the famous 'microdot', for example - to formal methods for obscuring the *meaning* of the message. This latter means of secrecy gave rise to the study of *cryptography* and the development of codes and ciphers. A code may substitute words or symbols for words and phrases in the original message or *plaintext*. A cipher acts on the characters contained in the message, using mathematical techniques (generally involving some combination of permutation and substitution) to create *ciphertext*, which one hopes will ensure secrecy of the underlying plaintext. The first formal cryptographic systems were mainly nomenclators, which were lists incorporating both codewords and cipher alphabets, usually for direct substitution.

As encryption techniques became more formalised, so did the need to develop techniques to reveal the original communication. *Cryptanalysis* is the term given to attacks on

cryptographic procedures in an attempt to recover the original message. The first major breakthrough in cryptanalysis was the development of frequency analysis by the Arabs around the end of the 12th century [64, chapter 2]. This may be used to exploit the natural redundancies of language to break the simple substitution ciphers of the day. Western cryptographers responded with the use of polyalphabetic ciphers in the early 15th century and the use of *keys* [64, chapter 3] to further strengthen the security of their communication, leading to the design of cryptographic procedures whose security depended solely on the key, rather than knowledge of the procedure itself (see Kerckhoffs [65]). The arrival of radio communication and the need for secrecy driven by world war prompted an unprecedented flurry of cryptographic activity in the late 19th and early 20th century. Cryptography was formalised and put into a more rigorous mathematical framework. The role of cryptography in many events having a huge impact on the history of mankind can be found in Kahn [64] and, more recently, Levy [79]. Details of both encryption and attacks using classical ciphers can be found in, for example, Fouché Gaines [42].

1.1.2 The impact of the computer

The arrival of well structured polyalphabetic cryptosystems such as that of Vignère (see, for example, Garrett [43]) during the 14th century arguably gave the cryptographer the upper hand for many years, although such ciphers were by no means invulnerable to experts employed in governmental 'black chambers'. The development and formalisation of mathematical methods gave hope to the cryptanalysts, but it was not until the development of machines such as Babbage's precursor to the computer in the 19th century that they managed to draw level once more. The concept of a brute force attack, hitherto unfeasible, was now made possible – a machine could check all possible decryptions, or certainly a large amount of them, in a matter of hours. Bletchley's attack on the famous Enigma machine during the Second World War provided an ample demonstration of the potential of machine-assisted cryptanalysis. As machines became more sophisticated, the time required to reveal messages grew shorter and shorter. On a modern computer, a simple polyalphabetic cryptosystem can be broken in a matter of microseconds. At the same time, a modern computer allows a cryptographer to perform a (hopefully) bewildering sequence of substitutions and permutations (such as that used by the *Data Encryption Standard* or DES [93] introduced in the late 1970's and only recently supplanted by the *Advanced Encryption Standard* or AES [96] in 2000), to the extent that we trust a machine to safely transfer our credit card numbers across the world with a high degree of security. One can in this respect say that cryptography plays a crucial role in the success of global e-commerce and is of vital relevance in the internet age. Indeed, as noted by Diffie [36], the development of the internet has had a similar impact on the development of cryptography as did the devel-

opment of radio some 100 years previously. The arrival of public key cryptography and procedures such as the RSA algorithm in the 1970s [37, 110] had enormous ramifications for cryptography, and opened up a wide variety of new possibilities for application of cryptographic techniques, allowing protocols enforcing not only confidentiality, but also message integrity, authenticity and non-repudiation.
There are a huge number of books available on modern cryptographic methods: see, for example, Garrett [43], Menezes et al. [87], Smart [123].

1.1.3 Attacks

The tactics of the cryptanalyst have obviously adapted to keep up with the developments in cryptography. One could perhaps say that the cryptologist currently has the upper hand, but this is not necessarily the case. Whilst algorithms may guarantee a certain level of security, or demonstrate the computational unfeasibility of an attack, it is another matter to make practical, secure protocols using these methods – see, for example, Anderson [6], Pfleeger [102]. The 'one time pad' method of encryption – combining the message with a random keystring having the same number of characters, which is then discarded and never re-used – is provably secure; but practical difficulties (such as generating and indeed protecting such keystrings) make it cumbersome and unwieldy for use in many situations.
We briefly consider here some common situations which can lead to attacks on cryptographic protocols and procedures. These may challenge either the mathematics behind the encryption process, the protocol, the implementation, or indeed any weak link in the entire chain. Of course, any cryptosystem is, in theory, susceptible to a brute force attack – one can try every possible key until the correct one is found. However, simply trying every key does not fully describe the amount of work needed to decrypt a given message, since one must also be able to actually detect a 'correct' decryption. This may not be easy if the original data had for example undergone some kind of compression prior to encryption. A good cryptosystem will make such a brute force search computationally unfeasible, even if one could harness the power of a huge number of computers – whether those of a single organisation, or via the internet in some open (or nefarious) collaboration.
We now consider various options which may be available to a potential attacker. An attacker may of course be active rather than simply a passive observer – in addition to trying to read messages, the attacker may try to delete, modify or replay messages in some communication protocol in order to meet his or her ends.

Ciphertext only

The bare minimum; also known as a passive attack. The attacker has merely a copy of the encrypted message. However, substitution ciphers and even polyalphabetic ciphers

may be broken with only this information, indicating their inherent weakness.

Plaintext–ciphertext pairs

Often called a lunchtime attack. The attacker now has access to the associated plaintext of a given ciphertext, and can attempt to establish links between the two. He may then make assumptions about the algorithm and attempt to derive the key. More sophisticated techniques such as *differential* and *linear cryptanalysis* use probabilistic methods to attempt to identify parts of the key of a symmetric cipher – this will hopefully reduce the search space for a brute force attack to determine the full key.

Chosen plaintext

The attacker may now obtain the ciphertext of any given message of his choosing. Public key cryptosystems, described in the next section, provide this information to an attacker by their nature. The attacker may now compare messages of a particular structure or make changes and view the effect on the ciphertext.

Man in the middle

The attacker here challenges the protocol rather then the encryption algorithm directly. One may attempt to take a copy of each message in a key exchange procedure, or one may substitute or modify the messages to change the outcome, or perhaps obtain the key or the contents of subsequent transmissions. For an example of this kind of attack, see Crouch and Davenport [32]. We note that an attack may simply reveal the contents of a single encrypted message, and not necessarily break the cryptosystem by finding the secret key. If, however, one can identify particular messages of high importance, this distinction may be moot.

Other

In certain situations, simply observing the amount of encrypted information being transmitted may provide useful information, even if it cannot be decrypted. This is known as *traffic analysis*, and was used successfully by the Allies in the first World War to indicate the commencement of some major enemy activity.
One can also attack hardware. One such method is *differential power analysis*, where one may attempt to identify power surges corresponding to the phases of an implemented algorithm, with a view to gleaning information about the message or key. For an example of such an attack, see Schindler et al. [111]. If the cryptographic hardware is not tamper-resistant in some way, one could modify the components to 'leak' information, or reverse engineer the equipment in order to gain further knowledge of the cryptographic procedure being used.

We may finally note that, in time-honoured fashion, a judiciously applied bribe or well-executed burglary may achieve the same effect as any of the above techniques; generally with considerably less effort than is required to challenge a well-implemented modern cryptosystem. Fortunately – perhaps – this is outside the scope of this thesis.

1.1.4 Public keys

Cryptography has come a long way in the past 25 years. Up until this time, ciphers had always relied upon some shared secret – a key – for their security. All ciphers were thus in some sense symmetric – what was done to encrypt was reversed to decrypt. In 1976, however, the idea of *public key* cryptography was first raised by Diffie and Hellman [37][1]. The basic idea of public key cryptography is that the decryption key is not the same as the key used to encrypt the original plaintext. This immediately goes some way towards resolving key distribution problems, but the ideas behind public key cryptography have been applied to many more applications than simply that of secrecy, as we now discuss.

Encryption

Public key cryptography allows a step away from the dependence on shared secrets or keys for secure communication. Separate keys are used for encryption and decryption, which are of course linked, but in such a manner that it is very difficult to derive the one from the other. This has triggered the search for so-called *trapdoor one-way functions* – a function which is extremely difficult to reverse without some special secret knowledge. One may argue that this still involves secret information; however the crucial point is that this secret information or trapdoor does not have to be disclosed to anyone. As the saying[2] goes, 'three people can keep a secret – if two of them are dead': not having to disclose the secret key essentially minimises the number of people one is required to trust. One member of the key pair may then be disclosed, in the hope that the amount of computation needed to derive the other makes such an effort unfeasible. Anyone may then use this 'public key' to send encrypted messages, but only the person holding the corresponding 'private key' will have the ability to decrypt them. Although significantly slower than symmetric key cryptographic algorithms such as the DES, the convenience of public key protocols has seen them become widely implemented for key exchange prior to further communication under a symmetric algorithm.
In the next chapter we shall take a closer look at the details of some public key protocols which depend for their security on the difficulty of computing logarithms over finite fields or other structures. Other procedures, such as the well known RSA algorithm of Rivest et al. [110], rely on the difficulty of factoring large numbers. In practice,

[1] This idea had been found by the British some 6 years previously, but the work remained classified.
[2] Attributed to Benjamin Franklin 1706-1790.

various protocols exist to register public keys and create so-called *public key infrastructures* (PKIs) to bind a user to a particular public key, leading towards the concept of electronic identity. Further modifications to protocols are required to deal with situations such as compromise of a private key, and indeed the creation and management of adequate keys requires a great deal of care – see the U. S. Government guidelines in [97]. The ideas can be taken still further, such that a participant's public key may be taken to be some function of their identity – for example, their email address or social security number (see, for example, Cocks [24]) – so that the need for public key directories no longer exists (although some other trusted service may still need to be employed). Since public key cryptography was first proposed in 1976, it has become a huge industry, particularly in the light of telecommunications advances and the growth of the internet with associated e-commerce.

Digital signatures

When one signs a paper document, or receives a signed document – for honest purposes – one makes certain assumptions. One would expect that the signature is 'bound' to the document in question in some way. Thus the document could not have been modified after it was signed. One would hope that one's signature could not be 'detached' in some way and used to fraudulently sign another document, and indeed that the signer of a document could not deny their signature at a later date. Such properties are achievable for digital signatures by making use of cryptographic protocols. The DSA or *Digital Signature Algorithm* as used in the *Digital Signature Standard* [94] attempts to satisfy the above requirements (basing its security on the discrete logarithm problem), as do various other protocols. Some make use of time stamps and message digests to reinforce the above criteria. As noted by Anderson and Needham [7], however, a certain amount of care is needed if one wishes to both sign and encrypt a message – signing after encryption may allow certain attacks to be made which allow the signature to be modified, replaced or even duplicated and applied to a completely different message. Digital signatures are of course very much linked to the concept of electronic identity, and thus one may make further use of public key cryptography in authentication protocols – at the most basic level, a host may store public keys and ask a user to encrypt a piece of information with their private key as a rudimentary password protocol.

Other

Many more esoteric protocols are made possible using cryptography; particularly public key cryptography. A public key algorithm can be made into a 'one-way' hash function by discarding the private key. Protocols exist for playing long-distance poker or flipping a coin over the internet. Proposals have been made for remote voting at elections,

for digitally certifying receipt of emails, and even for secure digital money. For a comprehensive description of a range of cryptographic protocols, see Schneier [114].

1.1.5 The state of the art

The rapid development of information technology and communication has driven equally rapid development in cryptographic techniques and, of course, associated attacks. In 1981 the American National Standards Institute (ANSI) approved the *Data Encryption Standard* or DES [93]; a cryptographic algorithm to be used as an industry standard. The DES remained the standard until 1998, but, with today's technology, has become vulnerable to a brute force search of its keyspace, either by dedicated hardware or by internet based parallel processing. In 2000 it was replaced by the new *Advanced Encryption Standard* (AES), a Dutch designed cipher called *Rijndael* [96]. Both the DES and AES are symmetric block ciphers – they work on blocks of plaintext rather than taking a character-by-character approach (known as a stream cipher). The security of the Advanced Encryption Standard can be increased by simply increasing key length. At the time of writing, the AES algorithm can use a key length of 128, 192 or 256 bits. Although the RSA algorithm has now been around for over 20 years it is still, with Diffie-Hellman, the primary public key algorithm. At the time of writing, an RSA key length of 512 bits is considered vulnerable, and a 1024 bit key or greater is the norm. These definitions depend on various factors. We may use asymmetric computing power, for example – we may encrypt using a smart card with limited processing power and decrypt on a large central processor. We may use one level of encryption for certain critical data, and trust a weaker level of encryption to secure less important information. One could argue that short-term data need not require as strong a key, in that an adversary has less time to mount an attack. This may be dangerous, however, in that short term 'secrets' may well have a disproportionately high value. A variety of designs for purpose built 'cracking machines' have been proposed – some plausible, others as yet theoretical – to the extent that a machine capable of breaking 1024 bit RSA is considered possible – assuming one has some serious money to spend (see Shamir and Tromer [117]). It may be noted that the level of paranoia and the suspicions of the user also play a heavy part in the choice of algorithm and key length.

Attacks on public key algorithms have led to the development of various techniques to calculate discrete logarithms (to challenge Diffie-Hellman and similar schemes) and to factorise large integers (to challenge RSA-type protocols). At present, the fastest known factoring algorithm is the Number Field Sieve, which has been used to factorise 'general' integers – those having no special structure – of over 512 bits. Using variants of this technique it is also possible to calculate discrete logarithms modulo a prime of over 400 bits. Factoring is, at this time, rather more advanced than discrete logarithm calculation (discrete logarithms of over 400 bits (120 digits) were first computed in 2001,

whereas the 129 digit number RSA-129 was factored in 1994), but, as the techniques used are very similar, it is possible that this discrepancy will narrow over the next few years. We will discuss the development of these and other methods in the following chapter.

The future...?

As noted by Ferguson and Schneier [41], the role of cryptographic protocols is to minimise both the number of people who need to trust one another, and the amount of trust they are required to have. At present, an enormous variety of cryptographic procedures are commercially available, of which a small number are generally accepted as providing adequate security; given that they are correctly implemented and use suitable parameters. It remains that the only provably secure system is the impractical one time pad. However, for key exchange purposes, a new method is in development which will apparently guarantee security. Quantum cryptography, as the name suggests, uses quantum mechanics and the properties of photons to transmit a key which would go some way to avoiding active attacks, due to Heisenberg's 'uncertainty principle' – any attempt to observe a quantum state inevitably alters it, so one cannot hope to eavesdrop on a communication without this intrusion being detected. Rumour has it that this has been successfully demonstrated across fibre optic cables over a distance of around one hundred miles, although adapting such a technique to transmit through a medium such as the atmosphere is another matter, due to the potential interference this will have on the transmission. It is as yet unclear whether such a key exchange mechanism will ever be a practical reality. To go still further into the realms of science fiction, the development of a practical quantum computer would potentially signal the end of most current cryptographic principles. Current opinion, however, seems to indicate that one should not lose sleep over the possibility.

1.2 Structure of the thesis

This thesis is concerned with computing discrete logarithms over finite fields using large prime variations of the well known index calculus method. As a result, the thesis is structured such that the main body of the work follows the order of the steps of the index calculus method itself. While this gives a logical direction to the sequence of chapters, it does however mean that a certain amount of forward and backward referencing is required. This is unavoidable, since choices that are made at different points in the procedure often do not yield benefits until further downstream. The thesis then divides, broadly speaking, into four parts.

1.2.1 Part I

Following this brief introduction to cryptography, in chapter 2 we review techniques and situations in cryptography which influenced the development of index calculus methods. We consider their application to the discrete logarithm problem (with an occasional glance at their application to factoring) and give an idea of the current 'state of the art' at time of writing. We introduce the basic ideas of the index calculus method and discuss various extensions and improvements to the procedure.

1.2.2 Part II

In chapter 3 we consider parameter choices for the index calculus method, and see how we can gain small but practical savings in both time and storage, for very little effort, by a judicious choice for the generator of the finite field under consideration. Although this is based on a very simple observation, such a technique has not been seen in the existing literature. We return to this technique in chapters 6 and 7 in order to view its overall effectiveness.

In chapters 4 and 5 we consider so-called 'large prime variants' of index calculus techniques, as applied to computation of discrete logarithms. In chapter 4 we describe an implementation of the index calculus method, examining the impact of the use of the single large prime and double large prime variants of this technique. We highlight the differences one needs to be aware of when applying the two large prime variation to discrete logarithm computation (as opposed to its application in factoring), and show how we can try to maximise the effectiveness of this technique in the discrete logarithm case. Our method of resolving such relations appears to be slightly simpler than those described in the few implementations of such techniques in the open literature. Again we must keep in mind later stages of the method, and we discuss how these techniques have a bearing on the work of chapters 6 and 7.

In chapter 5 we examine the use of the double large prime variation in conjunction with the 'Waterloo variant' of the index calculus technique, allowing us to use up to four large primes. We examine the effect of this technique, and investigate how the techniques of the previous chapter scale to resolve relations involving larger numbers of large primes.

1.2.3 Part III

Chapter 6 follows on from the methods in chapters 4 and 5 with an examination of the linear algebra step required by index calculus techniques both for factoring and discrete logarithm calculation. We examine the technique of structured Gaussian elimination and investigate the effects of using 'large prime' data on this method. We also return briefly to parameters used in the original relation generation procedure and see the

advantage we can gain from suitable generator choice as described in chapter 3. We then consider numerical methods, namely the Lanczos algorithm as applied over a finite field, and discuss the effectiveness of this technique when used in conjunction with structured Gaussian elimination for various inputs; notably those arising from the use of large prime techniques.
In chapter 7 we complete our computations by examining the final step of the index calculus procedure; that of actual evaluation of the logarithm of a given field element. We consider the impact of 'incomplete' data collection in phase 1 of the index calculus method, and subsequently examine how one may apply large prime variant techniques to substantially speed up this final part of the procedure.

1.2.4 Part IV

Finally, we summarise the results obtained in the thesis and discuss potential further work. We outline the main unresolved topics and make suggestions as to how these may be tackled in future. We subsequently give a brief history of the development of large prime variant techniques in appendix A, with a view to providing a comprehensive list of references for the interested reader.

1.3 Implementations

All procedures described in chapters 3 to 7 have been implemented in the C++ programming language, using version 5.0c of Victor Shoup's NTL number theory library [121] linked to version 3.1.1 of the GMP multiple-precision package, coupled with the GNU 'g++' compiler. Certain implementation techniques have also been taken from Loudon [84], Cormen et al. [29] and Jenkins [60]. These implementations were used to compute discrete logarithms of between 20 and 40 digits. For reference, we provide a 'map' of the major programs implemented in this study (along with the chapters where they are described) which can be referred back to if needed. The meaning of the labels will become apparent following the background given in the next chapter.

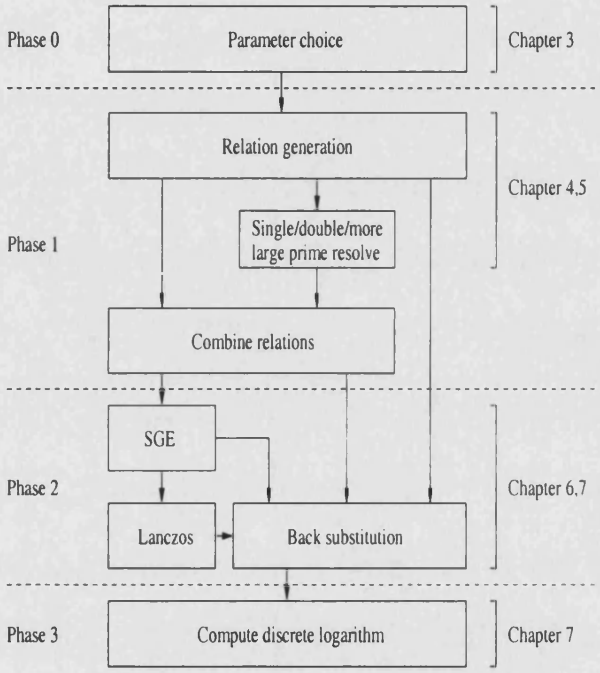

Figure 1-1: Overview of index calculus implementation

All code was run on a 1.8GHz Pentium IV machine, and all timings given are in hundredths of a second. While we have endeavoured to make code reasonably efficient, it has not been explicitly optimised by hand, and could probably be improved without too much effort. Further information concerning the datasets used in this thesis is given in appendix B.

Chapter 2

Research Background

In this chapter we introduce the discrete logarithm problem, and demonstrate how it may be used as the basis for key exchange and public key encryption. We then describe various methods for computation of discrete logarithms, with a final discussion of the ideas behind the index calculus method.

2.1 New directions

Let G be a group, and suppose $g \in G$ generates[1] some subgroup C_g of G. We define the **discrete logarithm** of $x \in C_g$ to be some value y such that

$$g^y = x$$

The discrete logarithm problem for G is then as follows: given $g \in G$ and $x \in C_g$, as defined above, compute y. The problem can be made still harder if one is simply given $x \in G$ rather than $x \in C_g$, since then we must firstly determine if a corresponding y even exists. Other versions of the problem exist where one has some further information, such as the Hamming weight of the discrete logarithm, or perhaps the knowledge that the discrete logarithm lies within a certain interval (see, for example, Teske [126]).
In this thesis, we take $G = (\mathbb{Z}/p\mathbb{Z})^*$ for p prime. Thus G is cyclic and has order $p-1$, g is now a *primitive root* of p, and every $x \in G$ has a corresponding discrete logarithm, taken to be the least non-negative integer y such that

$$g^y = x \bmod p$$

The discrete logarithm problem has been a focus for public[2] cryptographic research

[1] Recall that g is a generator of a cyclic group G if $g^{|G|} = e$ but, for all n dividing $|G|$, $g^{\frac{|G|}{n}} \neq e$ (where e is the group identity).

[2] 'Public' in the sense that the British government [46] and possibly others were aware of such methods several years earlier.

since 1976, when Diffie and Hellman first proposed the idea of public key cryptography [37]. Since this first key exchange protocol, a variety of methods for asymmetric encryption have been put forward. As noted in the previous chapter, the essence of public key cryptography is that the encryption key is not the same as the decryption key, and such a concept has many applications in addition to the original one (that of devising methods for secure communication). We note that the discrete logarithm problem appears in various other areas of computer science, and has uses other than the properties – namely its difficulty in many settings – exploited for cryptographic purposes. See Clark and Weng [23] for an example.

The first widely adopted public key cryptosystem was Rivest, Shamir and Adleman's 'RSA' [110] which was published in 1978, followed by a method of Rabin (see, for example, Smart [123]) a year later. The RSA algorithm draws its security from the supposed difficulty of factoring large numbers, whilst that of Rabin relies on the difficulty of computing square roots modulo some composite N of unknown factorisation. ElGamal [39] then introduced a public key cryptosystem based on the difficulty of the discrete logarithm problem in 1985. Further algorithms for encryption, digital signatures and authentication also rely on this supposed intractable problem. To see how discrete logarithms may be used for security protocols, we first consider the original key exchange procedure defined by Diffie and Hellman.

Suppose Alice[3] wishes to communicate with Bob under some symmetric encryption scheme. How do they exchange a key? Traditionally this was the realm of the trusted courier with a locked briefcase handcuffed to his arm; but Diffie and Hellman proposed the following protocol.

Suppose g is a generator of the group $G = (\mathbb{Z}/p\mathbb{Z})^*$ for some large prime p. The operator of the scheme used by Alice and Bob makes the values g and p public. Alice chooses some number $a \in G$ and computes $g^a \bmod p$. Bob picks some number $b \in G$ and computes $g^b \bmod p$. They now exchange these values and Alice computes $(g^b)^a \bmod p$. Likewise Bob computes $(g^a)^b \bmod p$. Since

$$(g^a)^b \equiv (g^b)^a \equiv g^{ab} \bmod p$$

they now share a value which they may use as a symmetric key. The security of this method depends upon the *Diffie-Hellman problem.*

Definition 2.1.1 (The Diffie-Hellman Problem). *Given a generator g, and given g^a and g^b, compute g^{ab}.*

An eavesdropping (but otherwise passive) attacker can discover the values g^a and g^b, and can obtain the public parameters g and p. If one could compute discrete logarithms

[3]In cryptographic literature, the two communicating parties are generally referred to as Alice and Bob. An attacker is usually referred to as Eve. There may of course be any number of users and attackers.

efficiently, one could compute either a or b and use this value to compute g^{ab}. The Diffie-Hellman problem is then at most as hard as discrete logarithm computation. It is currently an open problem as to whether the two problems are equivalent. A method for solving the Diffie-Hellman problem efficiently is not currently known, and for some particular groups it has been shown that the Diffie-Hellman problem reduces to the discrete logarithm problem (Maurer and Wolf [85]). In certain groups, it may in fact be difficult to compute any information about g^{ab} from g^a and g^b. This is the *Decision Diffie-Hellman problem.*

Definition 2.1.2 (The Decision Diffie-Hellman Problem). *Given the triples* (g^a, g^b, g^{ab}) *and* (g^a, g^b, g^c) *where the elements are in random order and* c *is some random group element, decide with probability greater than 1/2 which one is the correct triple, i.e. which is the triple* (g^a, g^b, g^{ab}).

As noted by Joux and Nguyen [63] and by Maurer and Wolf [85], the 'decision' problem is generally easier than the Diffie-Hellman problem proper. We note that the basic Diffie-Hellman key exchange protocol described above is vulnerable to a 'man in the middle' attack – in order to prevent an attacker modifying messages one needs to add some kind of authentication, such that Alice is assured that she is indeed communicating with Bob, and vice versa.

In addition to key exchange, one can define a simple discrete logarithm-based public key encryption scheme as follows. Suppose Alice wants to send a message M to Bob (securely) without sharing a symmetric encryption/decryption key. Alice thinks of a number a relatively prime to $p-1$. Bob thinks of a number b relatively prime to $p-1$. Alice now computes $M^a \bmod p$ and sends the result to Bob. Bob raises this message to the power b and sends the result, i.e. M^{ab}, back to Alice. Meanwhile, Alice computes a' such that $aa' \equiv 1 \bmod p-1$. She now sends Bob the result of the calculation $(M^{ab})^{a'}$; i.e. M^b. In order to read the message, Bob now computes b' such that $bb' \equiv 1 \bmod p-1$. Then

$$(M^b)^{b'} = M^{bb'} = M$$

and Alice has communicated with Bob without sharing a key. Of course, if one could compute discrete logarithms, one could find a and b directly.

Here we are of course considering discrete logarithms modulo a prime p. The discrete logarithm problem is not restricted to consideration over $(\mathbb{Z}/p\mathbb{Z})^*$, however; it can be generalised to other finite groups G. One may consider the above situation over, for example, the points of an elliptic or hyperelliptic curve over a finite field. Indeed, elliptic curves are attractive to the developers of discrete logarithm-based cryptosystems as one can get equivalent security to computation over $(\mathbb{Z}/p\mathbb{Z})^*$, yet using a shorter key length (due to the fact that the fastest algorithms for computing discrete logarithms over

$(\mathbb{Z}/p\mathbb{Z})^*$ do not generally apply to elliptic curve groups). This is important, as public key schemes are much slower than modern symmetric encryption algorithms. This is due mainly to the need for carrying out arithmetic operations on large numbers, rather than the bitwise substitutions and permutations employed in symmetric cryptosystems such as the AES. One generally uses a public key scheme for key exchange, before continuing communication under some symmetric encryption. Since AES keys are of 128, 192 or 256 bits, whereas at time of writing, public key algorithms such as RSA recommend the use of 1024 bit keys (or greater), this also keeps bandwidth down during communication. Certain cryptographic schemes make use of the discrete logarithm problem modulo some composite number N, such that the scheme is secure even if either factoring or discrete logarithm computation (but not both) ceases to become a computationally hard problem – see Girault [47].

Attacking the example of Diffie-Hellman encryption outlined above via calculation of discrete logarithms is illustrated as follows. Suppose that the malicious Eve wants to listen in on Alice's communication with Bob. If Eve intercepts a copy of the message at each stage in the protocol, and assumes that the original message M can be expressed as $g^n \bmod p$ for some generator $g \in (\mathbb{Z}/p\mathbb{Z})^*$, then on taking logarithms she has

$$\log(M^a) = \log((g^n)^a) = na$$

$$\log(M^{ab}) = \log((g^n)^{ab}) = nab$$

$$\log(M^b) = \log((g^n)^b) = nb$$

Using the fact that

$$\frac{nanb}{nab} = n$$

she can calculate $g^n \bmod p = M$ and the cryptosystem is broken. This attack, however, depends of course upon whether or not Eve can calculate the discrete logarithms required[4]. Again, the simplistic encryption scheme above is susceptible to 'man in the middle' attacks – the scheme of ElGamal [39] gets around this problem by use of an additional random session key which is never re-used.

In the ElGamal scheme, Alice's public key is (p, g, g^a), for p a prime and g a generator of $g \in (\mathbb{Z}/p\mathbb{Z})^*$, and her private key is the random integer a. A message M is represented as a nonzero element in the field GF(p). To communicate with Alice, Bob generates a random session key k and sends Alice the pair $(g^k, g^{ak}M)$. The ElGamal scheme thus suffers somewhat from message expansion. Alice then computes $(g^k)^a$ and uses this to retrieve M. Again, this basic description is modified in practice – see, for example,

[4]As a slight aside, it has been shown by Shor [119] that there exist polynomial time Las Vegas algorithms for both discrete logarithm computation and factoring for a quantum computer, should one ever be built. However, with current technology, both these problems remain intractable for suitably chosen parameters.

Smart [123]. The American 'Digital Signature Algorithm' (DSA) uses a variant of ElGamal's public key encryption algorithm [94], and discrete logarithms are used as the basis of security for various other cryptographic schemes. For a comprehensive survey of discrete logarithm-based cryptographic techniques, see McCurley [86], Odlyzko [98]. The discrete logarithm problem is, then, fundamental to the security of many modern cryptographic algorithms. Whilst discrete exponentiation is reasonably straightforward, reversing this operation has proved extremely difficult. Indeed, exponentiation modulo a large prime appears to be a true one way function – not simply a so-called trapdoor one way function as is the RSA function (the 'trapdoor' being knowledge of the prime factorisation of the modulus N). The importance of identifying algorithms and procedures for computing discrete logarithms is thus of great importance. A real-world example of an attack on discrete logarithm-based security is described by LaMacchia and Odlyzko [69]. Here, the Sun NFS[5] cryptosystem uses a modification of the Diffie-Hellman key exchange protocol as part of its authentication procedure. Each user and machine has a secret key m, and $g^m \bmod p$ is made public. A user passes the authentication test if he or she can prove knowledge of the secret key m. However, in the Sun system, the prime p and the generator g are (or rather, were) the same for every implementation of this software (p was in fact a 192 bit prime). Thus if one can calculate discrete logarithms, even if this takes a fairly long time (e.g. several months), then the security of the Sun system can be broken. It is even possible, as noted by Anderson and Needham [7], to make use of discrete logarithm computation to attack other protocols (the particular example being an attack on a message which has been encrypted and then signed using RSA).

At the time of writing, the recommended key size to secure data for the immediate future is 1024 bits (i.e. the recommended size for either an RSA modulus, or a prime p to be used in Diffie-Hellman or ElGamal-type schemes). The US KEA key exchange algorithm [95], which uses a Diffie-Hellman type protocol, recommends a key length of 1024 bits. Discrete logarithms have been computed over GF(2^{607}) by Thomé [127] in 2002, and over GF(p) for p a prime of some 400 bits by Joux and Lercier [61] in 2001. This apparent discrepancy is due to the existence of asymptotically faster algorithms, notably that of Coppersmith [26], for GF(2^n). On the side of factoring, a 512 bit RSA modulus was factored in 1999 by Cavallar et al. [22], giving proof to the opinion put forward several years previously that such keys could no longer be classed as secure for anything other than short term data. Van Oorschot [128] suggests that for security equivalent to factoring integers of 512 bits, we need a similar sized p – i.e. 512 bits in length – if using discrete logarithm based schemes over GF(p), but require 2^n to have some 700 bits if working over GF(2^n). In contrast, for elliptic curve cryptosystems (ECC), recommended key size is only 160 bits [97, page 85] – for an overview of ECC,

[5]Here 'NFS' refers to the Network File System.

see Koblitz [67], Miller [89], or Smart [123].
Attention has also turned to purpose designed 'cracking' hardware – see, for example, Lenstra and Shamir [76], Shamir and Tromer [117] – leading to the conjecture that a 1024 bit RSA modulus could be factored in less than a year by a machine costing some 10 million dollars. It is reasonable, given the similarity between the best known methods used for factoring and for discrete logarithm computation, to assume similar key sizes being required for discrete logarithm based schemes (such as Diffie-Hellman and ElGamal) as are recommended for RSA; although using elliptic curve based schemes would allow for smaller key sizes to be used for the same estimated levels of security.
Of course, when selecting a key one must bear in mind several points: the required duration of secrecy for the information, the permitted bandwidth for communication, the inherent value of the information itself, etc. The user of a cryptographic scheme basically wishes to make the effort required to attack a given communication greater than warranted by the value of the information itself. Choices for key sizes and suggested guidelines are discussed at length by Lenstra and Verheul [77] – since a variety of parameters affect the choice of key size, here the authors allow a user to estimate key sizes based on the user's opinion of the security of the DES algorithm. A different approach is taken by Silverman [122], where key sizes are analysed according to the value of the encrypted data and the cost of breaking a given key.

2.2 Computing discrete logarithms

The multiplicative group of of the finite field $\mathbb{Z}/p\mathbb{Z}$ is cyclic when p is prime, and is thus generated by a single element g. The order of the group is $p-1$, which, generally, is of course not prime. Any nonzero value x modulo p has a corresponding discrete logarithm y to the base g. Obviously, if x is the identity, or a small power of g, it is simple to find the corresponding logarithm, but in general, for large primes p, this is much more difficult. One could of course find such a y simply by computing

$$g, g^2, g^3 \ldots g^{p-1} \equiv 1$$

stopping when one finds a y such that $g^y \equiv x \bmod p$. This would take $O(p)$ attempts, and quickly becomes infeasible as p gets larger. Alternatively, storing all values and using some kind of hashed lookup would take $O(p)$ space.
In this section we take a look at some of the methods proposed for discrete logarithm calculation, following for the most part Garrett [43], Menezes et al. [87] and Smart [123]. After taking a brief look at the 'baby-step, giant-step' method of Shanks, Pollard's 'ρ' method, and the Pohlig-Hellman method, we consider the more specialised 'index calculus' ideas which ultimately created the Number Field Sieve (NFS), at time of writing the method of choice for factoring integers and computing discrete logarithms

at the limit of current expertise.

2.2.1 Shanks' baby-step, giant-step

The baby-step, giant-step method of Shanks [118] is a deterministic method which applies to any finite cyclic group. Here we give a brief description of the idea as applied to the discrete logarithm problem modulo a prime p.
Given a group G of order $n = p - 1$, generated by g, we can let $m = \lceil \sqrt{n} \rceil$ and then for some $x \in G$ we can say that $x = g^{i+jm}$ for some $0 \leq i, j \leq m - 1$. We can thus compute two lists: firstly the 'baby steps'

$$1, g, g^2, \ldots, g^{m-1}$$

which we store in a table such that we can easily look up a given value. We then compute the 'giant steps'

$$x, xg^{-m}, xg^{-2m}, \ldots, xg^{-(m-1)m}$$

and look for equality in the two lists, such that

$$g^i = xg^{-jm}$$

Then

$$x \equiv g^{i+jm} \bmod p$$

and thus $y = i + jm$ is the discrete logarithm of x.
Runtime for this method is $O(\sqrt{p})$, but it also requires $O(\sqrt{p})$ storage. We do not actually need to know n, the order of the group, since we simply need $m \geq \sqrt{n}$. If we choose some m and the algorithm fails, we can simply retry with a larger value for m. Modifications to this method are described by Buchmann et al. [18], where changes are proposed which lead to fewer 'baby steps' when $\log_g x$ is small compared to the group order $p - 1$.

2.2.2 Pollard's ρ method

The ideas behind Pollard's ρ algorithm [104] can be applied to both factorisation and discrete logarithm computation. It is similar to the 'baby-step, giant-step' method described previously, in that it also has expected runtime of $O(\sqrt{p})$; but has the advantage of requiring negligible storage. Now, however, we do need to know n, the order of the group. The ρ method is a probabilistic algorithm, and is currently the best known algorithm for computing discrete logarithms over the group of points of an elliptic curve (except for certain types of curve where the problem may be reduced to computing

logarithms in a finite field – see Menezes et al. [88]) or any other abstract structure. It is an example of a so-called 'Monte Carlo' algorithm – a randomised method which may produce an incorrect result, but for which the probability of this happening is bounded.

The original method partitions the group G of order n into three sets, denoted by S_1, S_2 and S_3. These subsets need a certain degree of care in their definition in that they should be of roughly equal size. In order to compute $\log_g x$ we create a sequence of elements y_i, where $y_0 = 1$ and

$$y_{i+1} = \begin{cases} xy_i & if \quad y_i \in S_1 \\ y_i^2 & if \quad y_i \in S_2 \\ gy_i & if \quad y_i \in S_3 \end{cases}$$

We note a further criteria on the choice of the partition of G – we do not allow 1 to be in set S_2. We then use this sequence of y_i to define two further sequences a_i and b_i, with $a_0 = b_0 = 0$ and

$$a_{i+1} = \begin{cases} a_i & if \quad y_i \in S_1 \\ 2a_i \bmod n & if \quad y_i \in S_2 \\ a_i + 1 \bmod n & if \quad y_i \in S_3 \end{cases}$$

$$b_{i+1} = \begin{cases} b_i + 1 \bmod n & if \quad y_i \in S_1 \\ 2b_i \bmod n & if \quad y_i \in S_2 \\ b_i & if \quad y_i \in S_3 \end{cases}$$

This defines a pseudorandom walk in the group. Notice that $y_i = g^{a_i} x^{b_i}$ for $i \geq 0$. We now make use of Floyd's cycle finding method – given (y_i, y_{2i}), we compute (y_{i+1}, y_{2i+2}) – in order to look for two elements such that $y_i = y_{2i}$. Using the fact that $y_i = g^{a_i} x^{b_i}$ and taking logarithms we get

$$(b_i - b_{2i}) \log_g x \equiv (a_{2i} - a_i) \bmod n$$

and, unless $b_i = b_{2i}$, and assuming we can compute $(b_i - b_{2i})^{-1} \bmod n$, we can thus determine $y = \log_g x$. If $\gcd(b_i - b_{2i}, n) \neq 1$ we repeat the procedure with a different value y_0.

It is possible to parallelise this method, at the expense of requiring a certain amount of central storage; see, for example, Smart [123] for details. Further improvements include a possible speedup of around 20%, brought about by using a different sequence y_i, as shown by Teske [125].

2.2.3 Pohlig-Hellman

As described by Pohlig and Hellman [103], in order to be effective, this algorithm[6] requires that the group order $n = p - 1$ has only small prime factors. The key result is that the discrete logarithm problem for some group G (generated by g, as before) is at most as difficult as the discrete logarithm problem for the largest subgroup of prime order in G. The Pohlig-Hellman method, then, reduces the runtime of the previous two methods to $O(\sqrt{q_i})$ where q_i is the largest prime factor of n, the order of the group. This provides a strong argument for choosing p for cryptographic purposes such that $p - 1$ has few small prime factors, in order that discrete logarithm computation must be carried out in at least one subgroup of order q where q is a large prime.
The Pohlig-Hellman algorithm proceeds as follows. Let p be an arbitrary prime where

$$p - 1 = q_1^{e_1} q_2^{e_2} ... q_k^{e_k}$$

is the prime factorisation of $p - 1$ and $q_i < q_{i+1}$. Given y, g and p, we are to find x such that $x = g^y \bmod p$. The algorithm determines y modulo $q_i^{e_i}$, and results are then combined via the Chinese Remainder Theorem (see, for example, Bach and Shallit [10]).
We decompose the problem of computing one discrete logarithm modulo p to that of computing e_i discrete logarithms modulo q_i, for each q_i where ${q_i}^{e_i}$ divides $n = p - 1$. The way we determine $y \bmod q_i^{e_i}$ is to expand thus

$$y \bmod q_i^{e_i} = \sum_{j=0}^{e_i - 1} b_j q_i^j$$

for $0 \leq b_j \leq q_i - 1$. We can then compute from the least significant coefficient 'upwards'. Suppose we are concerned with finding $y_i \bmod {q_i}^{e_i}$. Following Menezes et al. [87], we let $q_i = q$ and $e_i = e$ to simplify notation. Set $\gamma = 1$ and $b_{-1} = 0$, and let $\bar{g} = g^{n/q}$. We now compute the coefficients b_j, $0 \leq j \leq e - 1$. To compute b_j, first compute

$$\gamma = \gamma \alpha^{b_{j-1} q^{j-1}} \qquad \bar{x} = (x\gamma^{-1})^{\frac{n}{q^{j+1}}}$$

and then compute

$$b_j = \log_{\bar{g}} \bar{x}$$

using, for example, the method of Shanks or Pollard's ρ method. Once we are in possession of all the b_j, we hold the value $y_i = y \bmod {q_i}^{e_i}$. When we have solved for y modulo all values ${q_i}^{e_i}$ we can use the Chinese Remainder Theorem to combine the relations $y \equiv y_i \bmod {q_i}^{e_i}$ and thus find $y \bmod p$. Shoup [120] has shown that for an

[6] Sometimes referred to as the Silver-Pohlig-Hellman method.

arbitrary group G of order n, a runtime of $O(\sqrt{p})$, where p is the largest prime dividing n, is the best that can be hoped for from any algorithm which does *not* make use of special group properties. As noted by Peralta [101], as a result of the Pohlig-Hellman method, some low order bits of a discrete logarithm are easier to compute than higher order bits (giving rise to possibly 'indiscreet' logarithms).

2.2.4 Others

A variety of methods in addition to those noted above have been suggested for computation of discrete logarithms. These include others by Pollard such as his 'λ' or 'kangaroo' method [104], which requires that the logarithm lies within some known interval w. The method then has expected runtime of $O(w^{\frac{1}{2}})$, and has been shown by van Oorschot and Wiener [129] to be practical when used in conjunction with a Pohlig-Hellman decomposition to compute discrete logarithms which are known to be 'small'. For a comprehensive review of these methods and how they apply to various incarnations of the discrete logarithm problem, see Teske [126].
Many discrete logarithm techniques mirror techniques used successfully in factoring algorithms. At the time of writing, discrete logarithm calculation can be thought of as shadowing integer factorisation – however, many of the major breakthroughs in discrete logarithm calculation and in factoring have arisen from use of a method initially developed for the other application, suitably adapted in some way. Rather surprisingly, there exists a simple – but computationally useless – polynomial representation for discrete logarithms over finite fields by Wells, Jr. [135].

2.3 The index calculus method

The index calculus method has its roots in factoring, and the basic ideas of this method go back to ideas of Kraitchik [68, chapter 6] in the early 1920s. The method as applied to discrete logarithm computation first appeared in the work of Western and Miller [136], and was 'revived' by the renewed interest in factoring and discrete logarithm computation brought on by the advent of public key cryptography in the late 1970s and early 1980s (Adleman [1]). It can now take various forms; all however using the same basic ideas.
Hellman and Reyneri [53] note that if the discrete logarithm problem is easy for some non-negligible fraction of group elements, then it should not be much harder for others. If we wish to compute the discrete logarithm of some element x to the base g, we can multiply x by g^{r_i} for random values r_i until we reach a value where the discrete logarithm problem becomes 'easy'. The index calculus method looks to firstly build this set of 'easy' discrete logarithms, and subsequently uses them to compute arbitrary logarithms. The method is nondeterministic, and can be broken down into three phases.

These subprocesses within the index calculus technique have spawned further research into algebraic number theory and linear algebra.
Here we consider the basic index calculus method, as described by Adleman [1], and subsequently discuss various modifications which have been proposed both for factoring and for discrete logarithm computation.

2.3.1 Smooth numbers

Many algorithms, in particular those which have been proposed both for the computation of discrete logarithms over finite fields and for factoring integers, use the concept of *smooth numbers*[7], defined as follows.

Definition 2.3.1. *We define a number M to be* **B-smooth** *if all its prime factors are less than or equal to B, i.e.*

$$M = \prod p_i^{e_i}, \ p_i \leq B$$

One sometimes sees a number referred to as *B-rough* if it does not satisfy the above criterion. Estimates for the distribution of such numbers were developed well before the use of index calculus algorithms became common. These asymptotic estimates make use of the Dickman ρ function and the Prime Number Theorem as discussed by de Bruijn [34] and Hildebrand and Tenenbaum [56]. From Crandall and Pomerance [31, page 45], the probability that a number $x < y$ is B-smooth is approximately

$$u^{(-1+o(1))u}$$

as $u \to \infty$, where

$$u = \frac{\log y}{\log B}$$

Extensions to estimates for the distribution of smooth numbers can be found in Bach and Peralta [9] and Lambert [71], which discuss the distribution of so-called *semi-smooth* numbers having all but one or two prime factors less than some smoothness bound B_1, with the others themselves less than some other bound B_2. These results are further generalised by Cavallar [21]. A discussion of smooth numbers from a cryptographic point of view can be found in Wagstaff, Jr. [130, section 4.4].

2.3.2 Basic index calculus

The index calculus method, as mentioned, consists broadly speaking of three distinct phases – two of these comprise a precomputation step which is carried out once (and is the more time consuming), and subsequently a postcomputation phase allows a given discrete logarithm to be evaluated using information gathered in precomputation.

[7] Hardy and Wright [52] denote smooth numbers as 'round' numbers.

We wish to compute the discrete logarithm of some element $y \in (\mathbb{Z}/p\mathbb{Z})^*$ for p prime. Our first task is to select a subset of elements of G known as the *factor base.* At the most basic level, one takes the factor base P to be the k prime numbers up to a certain bound B. Thus

$$P = \{2, 3, 5, ..., q_k\}$$

The overall goal of the algorithm is to compute the discrete logarithms of these factor base elements (phases 1 and 2) and then to make use of these values in order to compute the discrete logarithm of any given group element (phase 3)[8]. In phase 1 of the method, then, we build a set of linear relations among the discrete logarithms of the factor base elements. There are several methods for accomplishing this: here again we consider the simplest method, as defined by algorithm 1.

Algorithm 1 Index calculus - phase 1

Input: Prime p, generator g of $(\mathbb{Z}/p\mathbb{Z})^*$, factor base P of k primes q_i less than some bound $B < p$

Output: Linear equations for discrete logarithms of (most) factor base elements

```
repeat
  Compute g^a mod p for random a ∈ (Z/pZ)*
  if g^a ≡ ∏ q_i^{e_i}, q_i ∈ P then
    Store a, e_i
  else
    Reject a
  end if
until #relations = k + ε for some ε
```

We compute $g^a \bmod p$ and check for smoothness. If this value is B-smooth, i.e. if

$$g^a = \prod q_i^{e_i} \bmod p, \quad q_i \in P$$

then

$$a = \sum e_i DL(q_i) \bmod p - 1$$

where the $DL(q_i)$ are the (as yet unknown) discrete logarithms of the prime factors q_i. Once we have enough of these expressions – some number greater than the number of factor base elements – we can solve them modulo $p - 1$ (or rather modulo the prime factors of $p-1$) to obtain values for the discrete logarithms of the elements of our factor base P. This linear algebra step comprises phase 2 of the method. We thus build a database of discrete logarithms of the factor base elements[9].

[8] We note that in many discussions of this method, one finds references to two phases rather than three; the step of solving the linear system is then the final part of phase 1, with computation now being phase 2. Here, however, we will *always* use a three stage model: relation generation in phase 1, linear algebra in phase 2 and subsequent computation of an arbitrary logarithm in phase 3.

[9] We shall sometimes refer to the process of relation generation as 'sieving', since most methods utilise some kind of sieve technique rather than the simple 'trial and error' method outlined here. It

In phase 3, in order to compute the discrete logarithm of an arbitrary x, and, assuming x is neither the identity, a small power of g, or itself B-smooth, we now pick a random a and compute xg^a. If this number is B-smooth, then

$$xg^a \bmod p = \prod q_i^{e_i} \bmod p, \quad q_i \in P$$

and so

$$DL(x) + a = \sum e_i DL(q_i) \bmod p - 1$$

We can now use our values for $DL(q_i)$ to compute $DL(x)$, as noted in algorithm 2.

Algorithm 2 Index calculus - phase 3

Input: Prime p, generator g of $(\mathbb{Z}/p\mathbb{Z})^*$, factor base P of k primes q_i up to some bound $B < p$, vector of discrete logarithms of factor base elements q_i, integer $x \in (\mathbb{Z}/p\mathbb{Z})^*$
Output: Discrete logarithm of $x \bmod p$
repeat
Compute $xg^a \bmod p$ for random $a \in (\mathbb{Z}/p\mathbb{Z})^*$
until $xg^a \equiv \prod q_i^{e_i} \bmod p, \quad q_i \in P$
Compute $y \equiv (\sum e_i DL(q_i)) - a \bmod p - 1$
Return $y = DL(x)$

2.3.3 Modifications

We now discuss the effectiveness of the index calculus method and note various modifications to improve its effectiveness. Some of these points will be expanded on in subsequent chapters of this thesis.

Relation generation

In addition to parameter choice, one can use a variety of methods to try to improve both the speed and the yield of the relation generation step. Many improved versions of the index calculus method try to reduce the size of the element that we test for smoothness; since a smaller number will generally be a product of a smaller number of smaller factors. This can be achieved in a number of ways – the simplest is to append the value -1 to the factor base. We can then subtract p from our value $g^a \bmod p$, and then effectively obtain two values to test for smoothness for a single exponentiation modulo p. We can opt to test both of these values or simply take the smallest in absolute magnitude. Taking this kind of idea further, one can introduce an intermediary level to the smoothness testing – one first expresses $g^a \bmod p$ as a product of 'medium sized' numbers, and subsequently tests these values for smoothness. This

was pointed out to us by a referee that this 'trial and error' method of generating relations is often denoted by 'Hafner-McCurley' variant index calculus, following the description of a similar procedure by Hafner and McCurley [51].

idea initially appears as part of the so-called 'Waterloo variant' of Blake et al. [13] (as described in a later chapter), and is taken further by the method of Coppersmith [26] for GF(2^n). Subsequent developments include the Gaussian Integer method (Coppersmith et al. [28]) which leads by way of the Cubic Integer method of Pollard [105] to the Number Field Sieve (Lenstra and Lenstra, Jr. (eds.) [73]) as used by Cavallar et al. [22], Schirokauer et al. [113], Weber [132]. Here, the chances of finding a smooth element are increased by testing two (smaller) elements over two factor bases, one algebraic and one rational[10].

Smoothness testing itself can be made more efficient in several ways. One can relax smoothness bounds by using so-called large prime variants of the index calculus method (Cavallar [21], Lenstra and Manasse [75], Leyland et al. [81], Odlyzko [98]), as investigated in later chapters of this thesis. One can use some kind of 'early abort' strategy if it seems unlikely that a given number will be smooth after having tested some proportion of the factor base elements, as outlined by Seysen [116], which can give a practical speedup of some 30% or more (Odlyzko [98]). Use of a sieve rather than simple trial and error effectively allows several values to be tested at once – see Coppersmith et al. [28], Pollard [106]. Further, rather than simply using trial division, as described above, the task of checking for smoothness itself can be improved in a variety of ways, as noted by Bernstein [12].

We note finally that relation generation is trivially parallelisable – we may simply test a different range of a values on as many different machines as we have available, as demonstrated by Lenstra and Manasse [74]. These may then be returned to a central repository for the linear algebra step. It is interesting to note the links between factoring and discrete logarithm computation, which can be seen in the development of the index calculus family of algorithms – the focus of the method switches from factoring to discrete logarithm computation and back again, as improvements are adapted from one setting to the other. Some of the above methods are restricted to particular settings, e.g. GF(2^n); however, Adleman and DeMarrais [2] present a version of the index calculus method for computing discrete logarithms in any finite field.

The linear algebra step

Index calculus methods call for the solution of a system of linear equations over a finite field. For factoring purposes, relation generation takes a fairly similar approach, but we then look for dependencies modulo 2 among relation exponents, with a view to finding an equation of the form

$$y^2 \equiv x^2 \bmod N$$

[10]Both factor bases may in fact be algebraic.

with $x \not\equiv \pm y$. We then hope to find a non trivial factorisation of our composite N by computing $\gcd(x-y, N)$. When computing discrete logarithms modulo a prime p, however, we must solve the linear system modulo $p-1$. In practice this often means solving modulo the prime factors of $p-1$ and subsequently using Hensel's lemma (if $p-1$ has repeated prime factors) and the Chinese Remainder Theorem (see, for example, Bach and Shallit [10]) to obtain a solution modulo $p-1$.

As mentioned previously, for cryptographic purposes p is often chosen such that $\frac{p-1}{2}$ is also prime, so that we are forced to solve modulo some large $q|p-1$. This has ramifications on memory requirements, and prohibits certain techniques which one may use modulo 2, such as storing 32 or 64 bitwise vectors in a machine word and processing all of these simultaneously. The structure of the matrix, however, remains much the same. Due to the nature of relation generation, most nonzero coefficients occur in the left hand columns, corresponding to the smaller primes in the factor base, since there is a $\frac{1}{q_i}$ chance that factor base element q_i divides a given value. Further, the matrix is extremely sparse, and most nonzero coefficients are ± 1 (with the exception of those corresponding to the smaller factor base elements, where coefficients may be 10 or more).

The linear algebra step turns out in practice to be a serious bottleneck, as it is considerably more difficult to parallelise than is the relation generation procedure. Methods such as that of Coppersmith [26] for $\mathrm{GF}(2^n)$ are asymptotically faster than the basic methods for $\mathrm{GF}(q)$, but for the precomputation phase this speedup occurs in relation generation – which is trivially parallelisable for all index calculus methods. It remains the case that a similar linear system must be solved. The only real advantage of the asymptotically faster methods such as NFS on the linear algebra step is that their increased speed in relation generation permits the use of a smaller factor base, leading to a smaller linear system. Nevertheless, the factorisation of the 512 bit number RSA-155 in 1999 by Cavallar et al. [22] required the solution of a linear system of some 6.7×10^6 rows and columns. The need to improve this step has influenced development and implementation of such techniques as structured Gaussian elimination, and the Conjugate Gradient, Wiedemann and Lanczos algorithms as applied to a finite field situation – see Cavallar [20], Coppersmith [27], LaMacchia and Odlyzko [70], Lambert [71], Lanczos [72], Montgomery [90], Pomerance and Smith [109], Wiedemann [137].

Computation

An important advantage of the index calculus method is that one only needs to run phases 1 and 2 once for a given finite field. This is a particular advantage if one considers for example a signature scheme with fixed parameters g and p – one can generate a database of small discrete logarithms, and subsequently run phase 3 (which is comparatively fast) to compute any discrete logarithm one chooses. If the parameters

are fixed indefinitely, one has the time to build such a database and attack signatures for the duration of the scheme. We note that techniques to speed up relation generation generally carry over immediately to phase 3, and also that we have in some sense a trade off concerning our factor base size – a larger factor base will decrease the runtime of phase 3 at the expense of solving a larger linear system, and thus increasing runtime and storage for phases 1 and 2. This has no analogy in factoring, where the only task after the linear algebra step consists of taking square roots of the values x^2 and y^2 and (hopefully) factoring N.

Runtime

It can readily be observed that precomputation will take considerably more time than postcomputation. The exact runtime of the procedure depends in a large part upon the choice of smoothness bound B. If we choose too large a B, our 'smoothness test' – such as trial division with the factor base elements – will become prohibitively slow. Conversely, too small a B will simply reduce our chances of ever actually finding sufficient a values such that $g^a \bmod p$ is B-smooth. Optimal choices for B are effectively best determined by a certain amount of trial and error coupled with implementation experience, but initial estimates can be given using smooth number probability estimates, which advocate taking

$$B = \exp(c\sqrt{\log p \log\log p})$$

for some constant c (McCurley [86]). Since the yield of basic relation generation is linear in the number of attempts made, it is simple to conduct short experiments using different smoothness bounds. This is also fairly straightforward for more complex methods of relation generation, although this may mean experimenting on subsections of some sieve interval.

Runtime for the index calculus method involves the function

$$L_p[\nu, \delta] = \exp((\delta + o(1))(\log p)^{\nu}(\log\log p)^{1-\nu})$$

If $\nu = 1$, then the function L is exponential in $\log p$, while if $\nu = 0$ then L is polynomial in $\log p$. For other values of ν, we are somewhere in between, and hence the runtime of the index calculus method is said to be subexponential. For variants of the Number Field Sieve, expected running time is $L_p[\frac{1}{3}, \delta]$ with $\delta = (\frac{64}{9})^{\frac{1}{3}}$ for the method as described by Schirokauer [112]. For the method in its most basic form as described previously, if we take $c = \frac{1}{2}$, runtime for precomputation is of order $L[\frac{1}{2}, 2]$ as $p \to \infty$. Phase 3, on the other hand, has an expected runtime of $L[\frac{1}{2}, \frac{3}{2}]$ (McCurley [86]). It is important to note that some of the strategies described earlier, such as early abort on smoothness testing, and the use of large prime variants, may not improve asymptotic runtime (but could give important practical speedup) – see Pomerance [107] for details.

Modifications such as the linear sieve can be shown to run in time $L_p[\frac{1}{2}, 1]$. Phase 3 for these methods has runtime of order $L[\frac{1}{2}, \frac{1}{2}]$ (see Odlyzko [98]).

2.4 Summary

In this chapter we have introduced the discrete logarithm problem, and given an overview of cryptographic techniques which use the supposed intractability of the discrete logarithm problem as a basis for their security.

We have given an overview of several well known methods for computing discrete logarithms modulo a prime p, and introduced the ideas behind the index calculus method. This method has evolved in different ways for different purposes. For discrete logarithm computation in GF(2^n), the best method known at the time of writing is that of Coppersmith [26]. For the more general case of computation in GF(p^n) for p a small prime, a newer development known as the Function Field Sieve (FFS) has been shown to be effective, with a complexity equal to that of the Number Field Sieve (see Joux and Lercier [62]). A further variation of Coppersmith's method for this situation is given by Semaev [115]. The method of ElGamal [40] can be applied to fields of the form GF(p^m) for fixed $m > 1$ and $p \to \infty$. For the case GF(p) for p prime, the fastest available method is the Number Field Sieve for discrete logarithms (see, for example, Gordon [48], Joux and Lercier [61], Weber [131, 132]).

Recent 'records' include computations in GF(2^{607}) using a version of Coppersmith's method (Thomé [127]), in GF(2^{521}) using FFS (Joux and Lercier [62]), and in GF(p) for p a prime of 400 bits using an NFS approach (Joux and Lercier [61]). Some implementations from this thesis are currently being used to compute discrete logarithms in fields of characteristic 3 via FFS (Granger et al. [50]). We note that such discrete logarithm computations take a serious amount of computing power – in the example given by Thomé [127], sieving took around a year on some 100 separate machines (mainly desktop PCs). The linear algebra step took around a month on a cluster of six 4-CPU machines. We cannot hope to match such computations, but can investigate computation techniques for smaller modulus sizes and attempt to estimate the benefits they may bring at 'cutting edge' sizes.

Part II

Index Calculus Methods

Chapter 3

Choice of Generator

Before our examination of large prime variant techniques, we firstly consider the advantage to be gained by making a particular choice of generator when using the basic index calculus method. We examine how one may make such a choice, and illustrate the benefits which may be obtained.

3.1 Changing bases

In the classical Diffie-Hellman key exchange protocol, g and n (where n is usually a prime p) are public. For ElGamal encryption [39], these and another parameter form the public key. Generally, for cryptographic purposes, one chooses a prime modulus p such that $p-1$ has at least one large prime factor q, and indeed one often chooses p such that $\frac{p-1}{2}$ is prime. In such a situation, p is known as a *safe prime*, and $q = \frac{p-1}{2}$ as a *Sophie Germain prime*[1]. Lim and Lee [83] advocate taking p such that $\frac{p-1}{2q}$ is also prime, or that each prime factor of $p-1$ is larger than q. They dub such a p a *secure prime*, since certain discrete logarithm based protocols can leak bits of the secret key in some situations, and such a choice for p helps to minimise this loss. If, on the other hand, one chooses p such that $p-1$ has many small prime factors, we allow the possibility of an attacker using the Pohlig-Hellman algorithm as described in the previous chapter. Since $p-1$ is usually composite, we can compute discrete logarithms modulo each prime factor of $p-1$ separately, and subsequently combine these values using Hensel's lemma and the Chinese Remainder Theorem (see, for example, Bach and Shallit [10]).

For security, it is then desirable for $p-1$ to have at least one large prime factor q, so that at least one subgroup has large order and we can thus force the attacker to take on what we hope to be a difficult computation. If this is not the case, we may allow the

[1] It is conjectured that there are an infinite number of such primes. Indeed, interest in this topic has recently resurfaced in connection with the new AKS deterministic polynomial time primality test of Agrawal et al. [4].

attacker to compute several discrete logarithms to smaller moduli, where more efficient methods may be applicable.

As an 'attacker' we do not have control over the choice of g made by the participants in a protocol, but we can be assumed to know what this chosen value is – it is, after all, a public parameter. Concerning this choice, we are told by Schneier [114] that there is no reason not to choose g to be as small a value as possible, such as 2. Indeed, van Oorschot and Wiener [129] note that choosing $g = 2$, coupled with a prime p for which $\frac{p-1}{2}$ is also prime, allows some computational savings during exponentiation when using Diffie-Hellman key exchange, or indeed most discrete logarithm based protocols. Further, Boneh and Venkatesan [17] recommend choosing $g = 2$ for a modified Diffie-Hellman scheme, leading to greater bit security in the ensuing shared key value.

We may note, however, that for the purposes of computing discrete logarithms over a finite field (for example, in an attack on the protocol), one may to a certain extent ignore the value of g used in the protocol. We may choose our own generator, g' say, and compute the discrete logarithm of a given x to the base g'. We may then use a simple change of bases to compute the discrete logarithm of x to the base g as

$$\log_g x = \frac{\log_{g'} x}{\log_{g'} g}$$

The complexity of the discrete logarithm problem is independent of the generator used – any algorithm used to compute discrete logarithms to the base g can also be used to compute to the base g'. Since to change bases in this way we need to compute $\log_{g'} g$, this does of course require the computation of two logarithms rather than simply one; but we may argue that the second phase of the index calculus method is of a sufficiently low comparative complexity that this increase in effort will hopefully be offset by any savings we could perhaps make in earlier phases of the procedure by using a generator of our own choosing. Further, in many situations, a public key pair (g, p) will be fixed (in, for example, a signature scheme), and thus we need only compute $\log_{g'} g$ once.

3.2 Reducing matrix density

So why might we want to use a different generator, and what should this choice for g' be? We note that, when computing discrete logarithms modulo a prime, we are always in possession of one discrete logarithm: that of the generator itself, since trivially

$$\log_g g = 1$$

Consider using the index calculus method as described in the previous chapter. If the given generator happens to be one of our factor base elements – and if, for example, $g = 2$ or $g = 3$, as is often the case, it will be – then we do not need to solve for the

discrete logarithm of this element in our linear algebra step. Rather, we can remove this element from any relation it occurs in as we build the matrix (taking care to adjust the 'right hand side' of the linear system), and thus remove an entire column – and hopefully a reasonable amount of the nonzero values – from the system prior to solving. It is simple to see that, in general, the column corresponding to the factor base element 2 will contain the most nonzeros – a given number $g^a \bmod p$ has a 50% chance of being even and thus containing 2 as a factor. In fact, we should have a greater than 50% chance of an entry in this column for a given row, since we know that the values $g^a \bmod p$ which lead to rows in the matrix are all B-smooth and are thus composed of small prime factors only. As a result, it is rather more likely that a row will contain a nonzero in the column corresponding to 2 than it would be for purely random values of $g^a \bmod p$.

This '2-column' will also generally contain the largest absolute values, since there is obviously more chance that some number $g^a \bmod p$ will contain a factor such as 2 repeated 10 or 12 times than it would a factor such as 101, for example, repeated by the same amount. Further, if we use large prime variants of the index calculus method, we must combine relations in order to eliminate the larger primes (as described in the next chapter), and this has a good chance of further increasing the absolute size of the matrix coefficients. As noted in Gordon and McCurley [49], and in chapter 6 of this thesis, whilst most values in the matrix are ± 1, the presence of these larger values can cause numerical blowup in the linear solve step, so any reduction in the amount of such values would also be beneficial from a practical point of view.

It is in our interests as an attacker, therefore, to have $g = 2$. From Crandall and Pomerance [31, section 2.5], it is conjectured that 2 is in fact a primitive root for infinitely many primes p. If 2 is the published generator, we need take no further action other than to remove the column corresponding to 2 in a preprocessing step prior to the linear solve procedure, or indeed on the fly during relation generation. We do not have to change bases in computation, and may proceed as normal. If 2 is not the published generator, we have a choice to make. If 2 is a generator of $(\mathbb{Z}/p\mathbb{Z})^*$ (but not, for some reason, the published generator) then we may use $g' = 2$, and proceed as before, but making a change of base in the final discrete logarithm computation. Should 2 not generate $(\mathbb{Z}/p\mathbb{Z})^*$, we would ideally like to choose a g' which will still allow us to remove the 2-column from the matrix prior to solving.

We also note that if we use the special value -1 in the factor base, we also hold the discrete logarithm for this value, since

$$\log_g(-1) = \frac{p-1}{2}$$

We can thus also remove the column corresponding to the factor base element -1 prior to the linear algebra step, as noted by Crandall and Pomerance [31]. We can expect a

nonzero entry to occur in this column in every other row.
In the next section, we assume that we are using the basic index calculus method outlined in the previous chapter. Thus we make use of some generator value in the relation generation step, and we consider how one could go about choosing a suitable value. In certain other methods, such as that of Coppersmith et al. [28], relation generation is in some sense 'generator independent'. We actually specify a base for the logarithms in phase three. For such methods, we could make the decision to compute to the base 2 in advance, and thus make savings in the matrix step. Again, we may subsequently have to compute one additional logarithm ($\log_{g'} g$) in order to change base.

3.3 'Useful' generators

The choices we could make for our generator g' may now be considered. We are looking to transform a relation such as

$$g'^{a} \equiv 2^{e_2} \times 3^{e_3} \times 5^{e_5} \times \ldots \bmod p$$

to a relation of the form

$$g'^{a'} \equiv 3^{e_3} \times 5^{e_5} \times \ldots \bmod p$$

Suppose that we choose a generator g' for which we know the value k such that

$$g'^{k} = 2 \bmod p$$

Then, for the previous example, we would have

$$2^{e_2} = (g'^{k})^{e_2} \bmod p$$

and so we would be able to remove the element 2^{e_2} by computing

$$g'^{a-(ke_2)} \equiv 3^{e_3} \times 5^{e_5} \times \ldots \bmod p$$

Is it always possible to find a k to enable us to remove the 2-column in this way? We assume that we have access to the prime factorisation of $p-1$ – this is not an unreasonable assumption, since the users of the protocol will probably make such information public to demonstrate the security of the system. We can thus quickly check if a candidate g' is indeed a generator (see Menezes et al. [87, section 4.6]). We wish now to show that one can always take either 2 or some root of 2 as the generator for $(\mathbb{Z}/p\mathbb{Z})^*$.

Following Garrett [43, chapter 13] and Li and Pomerance [82], we have

Theorem 3.3.1 (Euler's criterion). *Let p be a prime and let $k|\ p-1$ (so $p \equiv 1 \bmod k$). Let g be relatively prime to p. Then g is a k^{th} power residue modulo p if and only if*

$$g^{\frac{p-1}{k}} \equiv 1 \bmod p$$

Proof. If g is a k^{th} power then there exists some h such that $h^k \equiv g \bmod p$. Then

$$g^{\frac{p-1}{k}} = (h^k)^{\frac{p-1}{k}} = h^{p-1} \equiv 1 \bmod p$$

by Fermat's Little Theorem. Conversely, if $g^{\frac{p-1}{k}} \equiv 1 \bmod p$ then, letting g' be a generator of $(\mathbb{Z}/p\mathbb{Z})^*$, we have that $g \equiv g'^l$ for some l and so

$$g^{\frac{p-1}{k}} = (g'^l)^{\frac{p-1}{k}} \equiv 1 \bmod p$$

Since the order of g' is $\phi(p) = p-1$ by assumption, we thus have that $(p-1)|\ l(p-1)/k$. Then k must divide l, so $l = km$ say for some m. Thus

$$g = g'^l = g'^{km} = (g'^m)^k \bmod p$$

and g is a k^{th} power modulo p. □

For 2 to be a generator of $(\mathbb{Z}/p\mathbb{Z})^*$, it follows that 2 is *not* a k^{th} power modulo p for any k dividing $p-1$. Conversely, if 2 is *not* a generator of $(\mathbb{Z}/p\mathbb{Z})^*$, and instead has order m where $m|\ p-1$, then it is some k^{th} power of a field element, where $k = (p-1)/m$. This in turn will either be a generator or a power residue, and we can repeat the procedure (which will eventually terminate since we have a finite number of elements and $\phi(p-1) > 0$ are generators). If 2 is not itself a generator, therefore, some root of 2 modulo p is. We now consider how we can find such a root – simplicity of computation depends very much on the group structure of $(\mathbb{Z}/p\mathbb{Z})^*$, as this determines the value k. For background and further information on the rest of this section, see Garrett [43], Gauss [45], and especially Bach and Shallit [10, chapter 7].

3.3.1 Choosing g' when $p-1 = 2q$

We first consider the simplest (and probably the most common) case, where $p-1 = 2q$ and q is a prime. We have that $(\mathbb{Z}/p\mathbb{Z})^*$ has order $p-1$. We wish to find an element g', such that $g'^{p-1} \equiv 1 \bmod p$ and $g'^{\frac{p-1}{r}} \not\equiv 1 \bmod p\ \forall r|\ p-1$. If we can take $g' = 2$ then we need take no further action. If not, then suppose the order of 2 is $m < p-1$. By Lagrange's theorem we know that the order of a group element must divide the group order; that is, $m|\ p-1$. Since $m \neq 1$ and $m \neq 2$ (we ignore the case $p = 3$, since it is of

no cryptographic interest), we have that $m = q$. Thus 2 is a k^{th} power residue, where

$$k = \frac{p-1}{m} = \frac{p-1}{q} = 2$$

Since $p \equiv 3 \bmod 4$, we can find one square root of 2 by computing

$$a \equiv 2^{(p+1)/4} \bmod p$$

and the other by evaluating $b \equiv -a \bmod p$. The first of these, however, is the principal square root. This is itself a square; and is thus *not* a generator of $(\mathbb{Z}/p\mathbb{Z})^*$, so we take $g' = b$ with order $p - 1$.

Example

For a computation of discrete logarithms modulo a 40 digit prime $p = 10^{40} + 17407$, the smallest generator of $(\mathbb{Z}/p\mathbb{Z})^*$ is 5. Here $p \equiv 3 \bmod 4$ and $p - 1 = 2q$ for q prime, so we can take $g' = \sqrt{2} \bmod p$, which yields the two roots

$$a = 6971841133177927536427151032011471104081$$

and

$$b = 3028158866822072463572848967988528913326$$

We note that $b = -a \bmod p$. Here a is the principal root and is itself a square. As a result, it cannot be a generator for $(\mathbb{Z}/p\mathbb{Z})^*$, and indeed a has order $q = \frac{p-1}{2}$. We can, however, take $g' = b$, which has order $p - 1$ as required.

3.3.2 Choosing g' when $p - 1 = q\prod_{i=1}^{n} r_i$

The more general case is slightly more complicated. Suppose

$$p - 1 = q\prod_{i=1}^{n} r_i \qquad q \geq r_1 \geq \cdots \geq r_n$$

with the r_i not necessarily distinct. Again, since $2 \in (\mathbb{Z}/p\mathbb{Z})^*$, we know that the group element 2 will have order dividing $p - 1$. Since, by assumption, $q \gg r_i$, there is a good possibility that $2 \in C_q$, the cyclic subgroup of order q, or some higher order subgroup such as $C_{r_i q}$. We can in any case determine the order of 2 by computing 2^q, $2^{r_1 q}$, $2^{r_2 q}$ and so on, as we know the prime factorisation of $p - 1$. If, for example, $2^{r_j q} \equiv 1 \bmod p$ then we have that $2 \in C_{r_j q}$. As before, if 2 has order $p - 1$ then we can take 2 as a generator. Otherwise, the order of 2 is some product m of elements of $\{q, r_1, \ldots, r_n\}$, and 2 is a k^{th} power residue, where $k = \frac{p-1}{m}$.

We are thus required to compute a k^{th} root of 2 modulo p. Since $k \mid p - 1$ and is

probably not prime, this is not so straightforward as in the previous case. If $k = 2$, and $p \equiv 3 \bmod 4$ we can proceed as above. If $k = 2$ and $p \equiv 5 \bmod 8$, similar results hold. Otherwise, still with $k = 2$, we can try, for example, the probabilistic method of Tonelli (see Bach and Shallit [10]) which has expected runtime $O((\log p)^4)$. This method succeeds with probability $\frac{1}{2}$. Often, however, we will have $k > 2$. In this situation, we can employ the 'AMM' algorithm of Adleman, Manders and Miller [3]. Again we assume[2] we know the prime factorisation of $p - 1$.

Algorithm 3 AMM algorithm for d^{th} roots modulo p (p, d prime)

Input: Prime p, integers a and d (a is a d^{th} power in $(\mathbb{Z}/p\mathbb{Z})^*$)
Output: d^{th} root of a modulo p

```
Choose a random h ∈ (Z/pZ)*
if h^((p-1)/d) ≡ 1 mod p then
  Choose a different h
end if
Let p − 1 = d^s t where d ∤ t
(a_d, a_t) ← (a^t, a^(d^s))
g ← h^t
e ← 0
for i = 0 to s − 1 do
  Select e_i, 0 ≤ e_i < d, such that (g^(e+e_i d^i) a_d)^(d^(s−i−1)) = 1
  e ← e + e_i d^i
end for
d' ← d^(−1) mod t
(b_d, b_t) ← (g^(−e/d), a_t^(d'))
Choose α, β such that αt + βd^s = 1
b ← b_d^α b_t^β
Return b
```

This randomised algorithm (a generalisation of Tonelli's square root algorithm) accepts a *prime* divisor d of $p - 1$ and an integer a and returns the d^{th} root of a modulo p. It does however fail with probability $\frac{1}{d}$, and has runtime $O(d(\log p)^4)$. Bach and Shallit [10] recommend using this algorithm only for small values of d, namely values such that

$$\log d = O(\sqrt{\log p \log\log p})$$

As discussed earlier, for cryptographic purposes it is unlikely that p would be chosen such that $p - 1$ has many prime factors. As noted by Pomerance and Shparlinski [108], for almost all primes p, the multiplicative order of 2 modulo p is *not* smooth. As we assume $q \gg r_i$, we assume that there is a good chance that the order of 2 is at least q, and thus k will be very small compared to p; so in general we would expect to be within this bound.

[2] If for some reason the full factorisation of $p - 1$ is not available, all may not be lost - we can compute a gcd-free basis if we know a partial factorisation: again, see Bach and Shallit [10].

Another problem is that we want a k^{th} root, where k is probably composite. In order to get around this, we proceed as follows. Suppose the prime factorisation of d is

$$d = {d_1}^{\mu_1} \dots {d_j}^{\mu_j}$$

Let $x_0 = 2$. Then we successively solve

$$\begin{aligned} {x_1}^{{d_1}^{\mu_1}} &= x_0 \\ {x_2}^{{d_2}^{\mu_2}} &= x_1 \\ &\vdots \\ {x_j}^{{d_j}^{\mu_j}} &= x_{j-1} \end{aligned}$$

using AMM, to finally obtain $g' = x_j$, such that ${g'}^d = 2$ as required.

Example

Suppose $p = 10^{100} + 226617$. Now $p \equiv 1 \bmod 4$ and $p - 1 = 2^3 \times 3 \times q$ for a 99 digit prime factor q. The order of 2 modulo p is $2 \times q$, and so 2 is thus a k^{th} power modulo p, where

$$k = \frac{p-1}{2 \times q} = 2 \times 2 \times 3 = 12$$

We may then use AMM to compute a 12^{th} root of 2 modulo p. It turns out that, for this example, if one computes all twelve such roots[3], eight of these are generators for $(\mathbb{Z}/p\mathbb{Z})^*$. The other four 12^{th} roots are themselves cubes, and so are not generators. Taking g' to be one of these eight generators, then, we can remove the column corresponding to factor base element 2 from our matrix. Computing such a root takes very little time – for this example, to compute a single 12^{th} root took 1 hundredth of a second; whilst computing all 12 roots (and checking if they were or were not generators) took 7 hundredths of a second.

3.3.3 Computations in GF(2^n)

Before looking at the practical impact of these ideas, we briefly consider the index calculus method as applied to GF(2^n). From Menezes et al. [87], GF(2^n) is represented as a ring of polynomials over GF(2) modulo some irreducible polynomial $f(x)$ of degree n. The basic index calculus method proceeds much as before, but now our factor base consists of the set of irreducible polynomials of degree $\leq m$ where $m < n$. Here m is the equivalent of our smoothness bound B. Our factor base is thus

$$P = \{x,\ x+1,\ x^2+x+1, \dots\}$$

[3]We can accomplish this by computing a primitive 12^{th} root of unity modulo p - see Bach and Shallit [10, chapter 7].

We compute logarithms to the base $g(x)$, where $g(x)$ is a primitive element of GF(2^n). Zierler [138] shows how one can change from one representation of GF(2^n) to another in at most n steps, which suggests that we can again use a different representation to that published, if it is to our advantage to do so.

We have a slightly different weight distribution in the relations matrix than we had for computations modulo a prime. Now, x is the first element in our factor base, followed by $x+1$. There is a 1 in 2 chance that either of these will divide a given polynomial over GF(2) (again, since we are considering 'smooth' rather than random candidates, this is probably rather more likely than this). The matrix columns corresponding to these two factor base elements will be the heaviest in the system. It is to our advantage, then to take $g(x) = x$ or $g(x) = x+1$, in a similar manner to our wishing to take $g = 2$ previously.

Again, from Menezes et al. [87, page 163], we can compute a representation for GF(2^n) by generating a monic irreducible polynomial $f(x)$ of degree n over GF(2). Then GF(2^n) can be represented as $(\mathbb{Z}/2\mathbb{Z})[x]/f(x)$, and we can take $g(x) = x$ as a generator. As we needed the factorisation of $p-1$ in the previous examples, so now we require the prime factorisation of $2^n - 1$ here, but again we argue that such information should be available for cryptographic examples. We can thus choose a $g(x)$ such that we can remove *one* of the heaviest columns in the matrix; however, it does not seem that we can remove both these heavy columns. As a result, any savings will not be as pronounced as when working modulo p.

3.4 Theoretical savings

We briefly consider the theoretical savings we may expect to make by removing the 2-column (and the -1-column). Estimating the density of a row in the matrix for the basic index calculus method modulo p is related to estimating $\omega(n)$, the number of distinct prime factors of a positive number n. From Hardy and Wright [52, theorem 430], we have

$$\sum_{n \le x} \omega(n) = x \log\log x + B_1 x + o(x)$$

$$\sum_{n \le x} \Omega(n) = x \log\log x + B_2 x + o(x)$$

where $\Omega(n)$ is the number of prime factors of n including multiplicity,

$$B_1 = \gamma + \sum \left(\log\left(1 - \frac{1}{p}\right) + \frac{1}{p} \right)$$

$$B_2 = B_1 + \sum_p \left(\frac{1}{p(p-1)} \right)$$

and γ is Euler's constant

$$\gamma = \lim_{n\to\infty}\left(1+\frac{1}{2}+\cdots+\frac{1}{n}-\log n\right)$$

However, a key point is that such estimates generally concern *random* numbers less than some bound. Here we have the added criterion that the numbers under consideration are smooth, and are products of small primes only. Fortunately, Alladi [5] provides similar approximations for smooth numbers. Let $S(x,y) = \{n \leq x : p|n \Rightarrow p \leq y\}$. Then $\Omega(n)$ is approximated[4] by

$$\eta(x,y) = \log\log y + \mathrm{li}(\alpha\xi)$$

where $\alpha = \log x/\log y$ and ξ is the unique positive solution of $e^{\xi} - 1 = \alpha\xi$.
We can thus argue heuristically that, for average $n \leq x$,

$$\omega(n) = \log\log x + B_1 + o(1)$$

$$\Omega(n) = \log\log x + B_2 + o(1)$$

and so, for y-smooth numbers,

$$\omega(n) = \Omega(n) - \sum_p\left(\frac{1}{p(p-1)}\right) \approx \eta(x,y) - \sum_p\left(\frac{1}{p(p-1)}\right)$$

We can use this approximation to estimate the number of nonzeros in the average row of our matrix[5]. This gives the values in table 3.1, where B is the smoothness bound used.

Size of p	B	Nonzeros per row (estimated)	Nonzeros per row (actual)
10^{20}	60,000	7.81	7.74
10^{25}	100,000	9.18	8.93
10^{30}	300,000	10.02	9.82

Table 3.1: Average nonzeros per row for various p

For a 30 digit p, with smoothness bound B, we computed logarithms to the base 5. To accomplish this we solved a linear system of $m = 36,500$ equations in $n = 25,998$

[4]This approximation holds for $\log y > (\log\log x)^{(5/3)+\epsilon}$, although it can be extended to other ranges (see Hildebrand [55]).

[5]In chapter 5 of this thesis, we make use of a modified version of relation generation, where we test two values a and b (both $O(\sqrt{p})$) for smoothness, with the final relation coming from the computation of ab^{-1}. However, these two values are *not* independent - they are coprime. As a result, the above approximation gives an overestimate of the number of nonzeros in the average row.

unknowns. Using the approximation above, we expect some $e_r \approx 10$ nonzeros per row, giving $e = me_r = 365,000$ nonzeros in the full system. We can assume that the element -1 is represented in every other row, and that the element $g = 5$ is represented in m/g rows. In fact, it is likely that it will occur in more rows than this, since the numbers concerned are smooth rather than random. Using this assumption, however, we can approximate the number of nonzeros in the system after we have removed the columns corresponding to -1 and g as

$$e' \approx me_r - \frac{m}{2} - \frac{m}{g}$$

For the dataset under consideration, this corresponds to $e' = 339,450$. We can thus expect to reduce the weight of the matrix by some 7% by removing the column corresponding to -1 and that corresponding to $g = 5$. Had g been equal to 2, this would rise to 10%. As noted, this is probably an underestimate, due to the fact that the numbers under consideration are smooth.

3.5 Practical savings

We now look at some practical examples of the kind of savings we can make by choosing the generator such that we can remove the 2-column. We look firstly at the structure of the kind of linear systems produced by the index calculus method. In chapter 6 we shall examine the effect of reducing matrix weight when using various methods of solution.

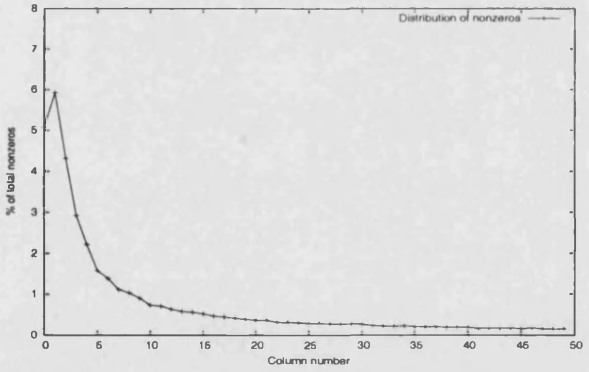

Figure 3-1: Distribution of nonzeros – 30b

Figure 3-1 refers to the first 50 columns of the $36,500 \times 25,998$ matrix mentioned above. The graph shows the percentage of the total amount of nonzeros which occur in each of the first 50 columns. In this example, the first column (index 0) corresponds to the value -1, which is often added to the factor base to speed up relation generation. A nonzero occurs in the -1 column for roughly half the relations, as one would expect. This column then contains 5.1% of the total nonzeros in the system. The second column

corresponds to the factor base element 2, and is clearly the column containing the most nonzero entries (around 5.9% of the total). In the fifth column (corresponding to factor base element 11), only some 1.5% of the total nonzeros occur. The 50th column contains some 0.16%. Of course, as we progress across the columns, many columns representing the larger factor base elements contain no nonzero values at all. By removing both the -1 and 5-columns for this dataset, we can remove 8% of the nonzeros from the system prior to solving. Had the generator been 2, this would have risen to 11%. Thus the model outlined above does indeed tend to underestimate the savings one can make.
This particular dataset was used to investigate the use of 'large prime' relations in discrete logarithm computation, as investigated in subsequent chapters of this thesis. Due to the fact that, by using this technique, we are combining several relations, the result is that the relations that we resolve are generally more dense. Consequently, we may expect the removal of the generator column to be less effective in reducing the overall density of the matrix. Considering figure 3-2 we see that this is indeed the case. Where previously 5.9% of the nonzeros in the matrix were in the 2-column,

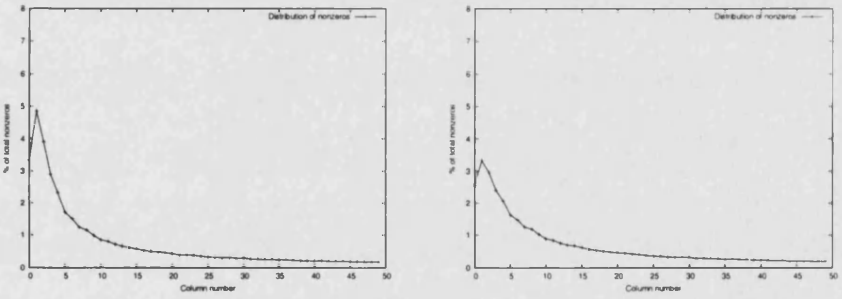

Figure 3-2: Distribution of nonzeros via large prime variant relations – 30b

when using relations derived using the 'single large prime' variant (left hand figure) this drops to 4.8%. When using relations derived using the 'double large prime' variant (right hand figure) this drops further to around 3.3%. On removing the -1-column and the 5-column now, we would save only 6.2% and 5%, rather than the 8% saved when using full relations only.
In general, if one were using these variants to build relations, one would still have a reasonable proportion – probably some 30-40% – of relations found directly, where the distribution of nonzeros would be similar to that in figure 3-1. The drop in effectiveness in removing the 2-column (from 5.9% down to 3.3% when using double large prime relations) would thus be diluted somewhat and not so pronounced in practice. On the other hand, linear algebra techniques for sparse linear systems are designed to avoid processing denser relations, so these relations derived from 'partials' may in fact be set aside before the costliest part of a linear solve method. We also note that the absolute size of coefficients in the linear system are generally larger when using relations derived

via large prime variants (see table 3.2 for coefficients in this 30 digit p dataset), so removing a certain amount of these would be beneficial to reduce or delay the effects of any coefficient blowup in pivoting. Indeed, it may increase the range for which 32-bit values may be used to represent data values without numerical breakdown occurring.

Relation type	Max in 2-column	Max in 3-column	Max in 5-column	Max in 7-column
Full relations	19	14	7	6
Via 1-partials	18	12	8	7
Via 2-partials	27	18	10	8

Table 3.2: Maximum nonzero values in first matrix columns – 30 digit p

This example is obviously rather small. As we scale up to larger datasets, we may expect that again the effectiveness of this column removal will be reduced, since again there will in general be more factors in a given relation. Indeed, for a 20 digit p dataset with $g = 2$, we were able to remove nearly 9% of the nonzeros occuring in the 2-column alone. Further, larger implementations may be expected to use more complex methods of relation generation than the simple 'trial and error' method described above.
The dataset in figure 3-3 was derived from an implementation of the Number Field Sieve (NFS) by Damian Weber [134] to compute discrete logarithms modulo a 90 digit prime p. Here

$$p = 120^{44} - 13 \qquad p - 1 = 2 * q$$

and q is prime. The matrix has 201,148 rows and 201,111 columns. The NFS, for the purposes of this discussion, uses two separate factor bases. For this particular implementation, one consists of rational primes as above. The other consists of prime ideals, and we look for relations involving both rational and algebraic numbers. One may thus argue that the effectiveness of removing the column corresponding to the rational prime 2 will immediately be halved, but in fact it is not necessarily that bad. One often chooses skewed parameters, such that the algebraic factor base contains fewer elements than the rational factor base (or vice versa). In this example, there were 7000 algebraic factor base elements and 15,000 rational elements.
Figure 3-3 shows the proportion of nonzeros occurring in the first 50 matrix columns corresponding to the rational factor base elements. On the left, they are shown proportionate to the total number of nonzeros in the whole system, and on the right they are shown proportionate to the total number of nonzeros which occur in the columns corresponding to the rational factor base elements only. In the left hand figure, we see the last few 'algebraic' columns contain a negligible (although actually nonzero) percentage of the total nonzero entries. The first rational factor base element here is 2, and corresponds to column index 7001. Subsequently column index 7002 represents

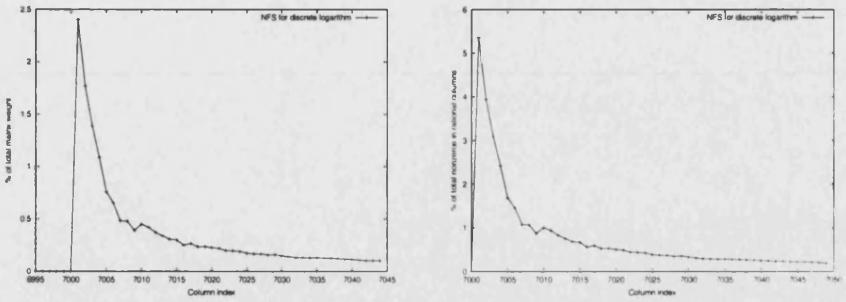

Figure 3-3: Distribution of nonzeros in NFS rational factor base

prime 3, and so on. We see that the 2 column contains some 2.4% of the total nonzeros in the system. This corresponds to 71,908 of the 2,966,157 total nonzero values, which we could remove from the system had we chosen our generator in a suitable manner (as it happens, g was actually equal to 2 for this example).

The right hand figure allows us to examine the importance of the 2 column as a proportion of the nonzeros occurring in the rational columns only. It actually accounts for some 5.4% of these values. This is obviously less than the 7% we had before, as our relations are going to have more nonzero values (relations in this dataset contain on average 15 nonzeros, compared to some 9.7 for the 30 digit example), although these nonzeros are going to be spread across both factor bases. All the same, by removing this column we can save some 800KB memory in storing this system of equations (assuming two long data types and one pointer per nonzero). Unlike the techniques described in Cavallar [20] and Huizing [59], where one is allowed to remove the k heaviest columns of the matrix prior to solving (further solutions are subsequently recovered), we do not have to build this column back in at a later stage of the solving procedure.

3.6 Summary

We have shown that it is possible to choose the generator used in the index calculus method such that one can subsequently remove a certain proportion of nonzero values from the ensuing matrix and hopefully speed up the linear algebra step of the method. This latter point will be investigated in a later chapter. Removal of the column corresponding to factor base element 2 is most beneficial, as a result of both the number of nonzeros contained in this column, and their absolute size. We have shown that we can always choose a generator which will allow us to remove this column, and, in general, we should be able to compute this value with little effort for the kind of primes p used in cryptographic schemes.

The effectiveness of this trick is diluted when using NFS-type approaches or large prime variant modifications to the basic method, and, of course, if we have changed the

published generator we will require a second discrete logarithm computation in order to change bases. We would hope that savings in the linear algebra step outweigh the effort of this second computation – again, this will be investigated in a later chapter.

Chapter 4

Large Prime Variants

In this chapter we examine the effectiveness of so-called large prime variants of the index calculus method as applied to discrete logarithm computation modulo a prime p. We first briefly consider the single large prime variant, followed by the double large prime variant as was applied to factoring by Lenstra and Manasse [75]. Although it is a natural step to take, there is very little discussion of the double large prime technique for discrete logarithm computation in the literature – in contrast, the technique is described in many papers concerning integer factorisation. However, Thomé [127] and Weber [132] show that others had also considered and implemented variations of this technique; although few details are given. The following work was carried out entirely independently of the results contained in these references.

4.1 Large primes in relation generation

In this section we discuss the single and double large prime variant, focusing on the differences between their application in factoring and discrete logarithm applications. Our intention is to examine how the double large prime method in particular differs when applied to discrete logarithm calculation, and how best to circumvent any practical complications which may arise. An overview of the history of these techniques and further references – mainly from a factoring point of view – can be found in appendix A.

4.1.1 One large prime

As described by Lenstra and Manasse [75][1], a simple modification to the basic index calculus method can be made. Although the authors apply this modification to factorisation problems, it is equally applicable to the discrete logarithm problem. In an attempt to speed up the relation generation phase of the index calculus method, we effectively relax our chosen smoothness bound B as follows. Instead of restricting our-

[1]This reference is not the first implementation of this technique – see appendix A.

selves to gathering relations only when $g^a \bmod p$ is smooth, we now retain relations involving one prime larger than our first smoothness bound B_1 (but below some second bound B_2). We shall refer to such relations as 'partial' relations. It makes sense to choose $B_2 < {B_1}^2$, since then, after division by factor base elements, if the remainder is less than B_2 then it must be prime.

If we discover two partial relations having the same large prime, we can eliminate the large prime and recover a full[2] relation. For factoring applications this can be accomplished by a simple multiplication of the two relations, and the large primes then disappear modulo 2. For discrete logarithm computation modulo p we are working modulo $p-1$, but it is equally simple to divide one relation by the other to eliminate the large primes. As we were generating (but rejecting) these partial relations anyway during the course of the standard method, we reduce the number of 'test values' needed to produce the necessary number of relations. A degree of complexity is of course added by the need to eliminate the large primes; but in practice this should be offset by the reduced time needed to gather full relations. Solving the linear system is then carried out as for the basic method.

4.1.2 Two large primes

It is not difficult to envisage going a step further and developing a 'two large prime' variation. Using two large primes is, however, not as straightforward as allowing one large prime only. Details of how this was first carried out (as applied to factoring) can be found in Lenstra and Manasse [75]. We first define a third smoothness bound B_3. We then modify the basic index calculus algorithm, defined previously, thus[3]:

- If $g^a \bmod p$ factors completely using only elements from the factor base (i.e. it is B_1-smooth), store as a full relation.
- If $g^a \bmod p$ factorises but for one larger prime factor Q with $B_1 \leq Q < B_2$ (i.e. it is 'semi-smooth'), store as a 1-partial relation.
- If $g^a \bmod p$ factorises but for two larger prime factors Q_1 and Q_2 with $B_2 < Q_1 Q_2 < B_3$ and $B_1 < Q_1 \leq Q_2 \leq B_2$ (i.e. it is 'doubly semi-smooth'), store as a 2-partial relation.

Again, a judicious choice of bound will help us here. Taking $B_3 \leq B_1^3$ guarantees that a composite remainder $R \leq B_3$ will have exactly two large prime factors Q_1 and Q_2 where

$$B_1 < Q_1 \leq Q_2 \leq B_2$$

[2]Following Lenstra and Manasse [75] we will refer to a relation involving purely factor base elements as a 'full' relation. From now on, we will distinguish partial relations by referring to them as '1-partials' for those having a single large prime, '2-partials' for those having two large primes, and so on.

[3]A brief word on notation – here we use q_i to represent factor base elements and Q_i to represent large primes above our smoothness bound.

In practice, one generally takes a substantially lower bound than this in order to reduce the amount of data generated. Typical values for B_3 are between $10B_2$ and $100B_2$, as noted by Boender [15]. However, we still have the added criterion that we must firstly check that the remainder is both below B_3 *and* is composite (the latter can be checked using the Miller-Rabin probabilistic primality test[4]), and subsequently we must factor this remainder in order to determine the exact value of these large primes. For the purposes of this implementation we chose to follow Lenstra and Manasse [75], and used Pollard's ρ method. For factoring, the rho method proceeds as detailed in algorithm 4. Here we show the basic algorithm; it can be made more efficient, as described by Knuth [66].

Algorithm 4 Pollard ρ for factoring

Input: Composite integer N
Output: n, a non-trivial factor of N

$x \leftarrow 5$, $y \leftarrow 2$, $k \leftarrow 1$, $l \leftarrow 1$, $n \leftarrow N$, $c \leftarrow 1$
loop
  Compute $g = \gcd(x - y, n)$
  if $g = 1$ **then**
    Advance:
    $k \leftarrow k - 1$
    if $k = 0$ **then**
      $y \leftarrow x$, $l \leftarrow 2l$, $k \leftarrow l$
    end if
    $x \leftarrow (x^2 + c) \bmod n$
  else if $g = n$ **then**
    Failure. Restart with $c = c + 1$
  else
    Divide out gcd:
    $n \leftarrow \frac{n}{g}$, $x \leftarrow x \bmod n$, $y \leftarrow y \bmod n$
  end if
  If n is prime, return n, $\frac{N}{n}$
end loop

In the case of failure, the algorithm would return the initial number as a factor of itself. This trivial result necessitates restarting the algorithm with a different value of c in the initialisation. In practice, for our implementations, on the few occasions that this situation arose, a correct factor was always obtained at the second attempt. We note that the ρ method is not necessarily the best option for factoring the remainder – it obviously depends on how large this remainder is. As datasets get larger, it would probably be more effective to use another method, such as Lenstra's elliptic curve factoring method (see, for example, Cavallar [21], Stephens [124]).

[4] On occasion, this test can flag a composite number as prime (see Davenport [33]) – for our purposes, this means we will reject a relation which would actually be correct. This failing will however have no bearing on the subsequent correctness of the algorithm.

The use of large prime relations does not change the asymptotic complexity of the index calculus method – it only changes the o(1) value, as shown by Pomerance [107]; but does give useful speedup in practice. However, in addition to the increased load on the relation generation algorithm entailed by this factorisation, we also need a more sophisticated method in order to resolve further full relations from the collection of 2-partials this will create, as we now investigate.

4.1.3 Resolving partial relations

To resolve partial relations we must eliminate the large primes in order to ensure that all relations consist only of elements from the factor base. As we have noted, for single large prime relations, or '1-partials', we can do this simply by identifying two relations involving the same large prime Q, and then dividing the one by the other. In practice, of course, one does not strictly divide the two relations; we simply subtract the exponents in the factorisation of one from those of the other. If we find k 1-partials with the same large prime, we can resolve $k-1$ full relations from these. Resolving 1-partials, then, is simply a matter of storing them in some ordered way, such as by use of a hash table, storing the relations according to the hash of their large prime.

Resolving 2-partial relations is rather more complex. Lenstra and Manasse [75] apply the two large prime method to integer factorisation with a quoted speed-up factor of 2–2.5. Relations involving two large primes are resolved by use of a graph algorithm. The large primes are represented as the vertices of a graph, and a relation involving two large primes is considered as an edge in this graph[5]. For the purposes of factoring some integer N, we are looking for relations of the form $x^2 \equiv y^2 \bmod N$, and so a cycle in the aforementioned graph is sufficient to 'eliminate' the large primes from a relation, since each large prime will appear an even number of times in a cycle. Resulting 'full' relations may then be solved modulo 2 (and as the large primes occur an even number of times, they vanish), and then $\gcd(x-y, N)$ is a non-trivial factor of N. Again, the situation is slightly different for discrete logarithm calculation modulo p, since we are *not* working modulo 2 – it is not sufficient that large primes occur an even number of times, since we must eliminate large primes modulo $p-1$. Weber [132] [134] accomplishes this by firstly constructing cycles in the graph, and subsequently building and solving a small linear system to eliminate the large primes. He reports that in some cases this procedure failed to resolve up to 2.8% of the cycles found. We now look at the nature of the cycles in the graph, in order to see if we can avoid this latter step.

Consider figure 4-1, which shows an even and an odd cycle. The Q_i correspond to large

[5]To clarify our terminology in comparison with that found in standard texts, our graph is allowed to have loops, but in this chapter we assume that there are no multiple edges - these are removed and processed separately in the same manner as single large prime relations.

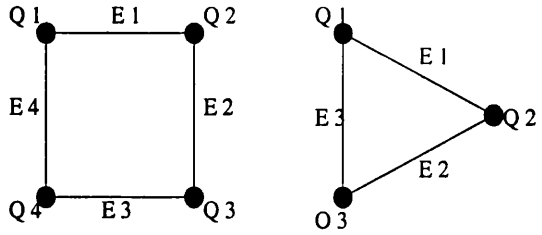

Figure 4-1: Using graph cycles to eliminate large primes

primes and the E_i correspond to relations such that, for example, E_1 represents the relation

$$g^{a_1} = Q_1 Q_2 f_1$$

where f_1 is the factorisation using elements from the factor base only. For the even cycle on the left, evaluating $(E_1 \times E_3)/(E_2 \times E_4)$ leads to a relation of the form

$$g^{a_1 - a_2 + a_3 - a_4} = \frac{Q_1 Q_2 Q_3 Q_4}{Q_2 Q_3 Q_4 Q_1} \frac{f_1 f_3}{f_2 f_4} = \frac{f_1 f_3}{f_2 f_4}$$

thus eliminating the large primes Q_i. We can then avoid the need to solve a linear system, and can instead eliminate large primes as the cycles are built. A certain amount of care is of course needed in processing cycles, so that we ensure all large primes cancel. Such an operation cannot be carried out on the odd-length cycle on the right without one of the large primes remaining – for example, computing $(E_1 \times E_3)/E_2$ leads to a relation of the form

$$g^{a_1 - a_2 + a_3} = \frac{Q_1 Q_2 Q_3 Q_1}{Q_2 Q_3} \frac{f_1 f_3}{f_2} = {Q_1}^2 \frac{f_1 f_3}{f_2}$$

We cannot eliminate all the large primes in the cycle. This implies that we must look for *even* cycles in our graph, since on first sight we cannot eliminate the large primes in an odd cycle. The way around this problem is to make use of an additional 'special vertex' 1. If this special vertex appears in an odd cycle, we see that we may effectively remove all the 'large primes' if we could order the edges such that our Q_1 in the example above was actually the vertex 1. Adding 1 to the graph is accomplished by treating the 1-partial relations as pseudo 2-partial relations (having 1 as their second 'large prime'). We may then hope to make use of a certain proportion of the odd cycles in the graph and improve our overall yield. As it happens, adding 1 to the graph is standard practice (see, for example, Atkins et al. [8]), as it allows us to process both 1 and 2-partials together in the graph. As we shall see, this leads to a variety of benefits, including higher yield and shorter cycles. We now discuss how to build and resolve these cycles, following for the most part Lenstra and Manasse [75].

A set of fundamental cycles forms a basis for the set of all cycles in the graph. From

elementary graph theory, we have that

$$\#\text{fundamental cycles} = e + c - v$$

where e, c and v are the number of edges, components and vertices in the graph, respectively. There will generally be many different possible sets of fundamental cycles. If we were restricting ourselves to looking for even cycles only, some sets may be more useful than others. Consider, for example, figure 4-2. This simple graph has 5 edges, 4 vertices and 1 component, and thus 2 fundamental cycles. It can clearly be seen that the graph contains three cycles – two odd and one even – but any one can be constructed from the other two. Our set of fundamental cycles may be made up of one odd and one even cycle, or possibly two odd cycles; in which case further processing would be required to recover the even cycle.

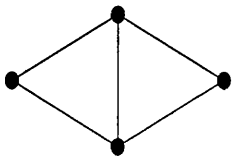

Figure 4-2: Simple graph with 2 fundamental cycles

Various algorithms exist for determining a set of fundamental cycles of a graph. In our implementation we again followed the approach of Lenstra and Manasse [75], which essentially breaks into two phases. The first phase is used to determine the structure of the graph. The second phase then builds a set of fundamental cycles. Given a set R of 2-partial relations, we assume that we know e, the number of relations in the set, and we assume further that the set contains no duplicates or incorrect relations which may arise due to corrupted data (if using distributed sieving, for example). In order to determine the number of fundamental cycles in the graph we must thus determine c and v. We can do this via algorithm 5. This uses a hash table T large enough to accommodate all large primes in R. Each entry in the hash table stores a value d_i and an ancestor value a_i.

The values of e, c and v can then be used to compute the number of cycles in the graph, and so evaluate the progress of the relation generation step of the index calculus procedure. With regards factoring, as mentioned, any cycle will yield a full relation. Were we looking for even cycles only, we could make a naïve estimate given that roughly half the fundamental cycles will be even in length (although as we will show, we can in fact do considerably better than this in practice). Once we have determined the number of fundamental cycles in the graph, and that no more partial relations are required, we must go ahead and build a set of fundamental cycles. We do this by means of a breadth-first search of the graph as described in algorithm 6. Once again we make use

Algorithm 5 Determine number of fundamental cycles of a graph

$c \leftarrow 0$, $v \leftarrow 0$, $d_i \leftarrow -1 \forall i$
Insert vertices Q_1 and Q_2 into the graph:
Compute index $j = h(Q_1)$ for some hash function h
Lookup in hash table:
if $d_j = -1$ **then**
  We have a new prime in the graph
  $d_j \leftarrow Q_1$, $a_j \leftarrow j$, $v = v + 1$, $c = c + 1$
else if $d_j = Q_1$ **then**
  Vertex has already been processed. Ignore
end if
Repeat for Q_2
Find the roots r_i of the components to which Q_1 and Q_2 belong
To find the root for vertex Q_1 (where $d_j = Q_1$):
$r \leftarrow j$
Recursively set $r \leftarrow a_r$ as long as $a_r \neq r$
When $a_r = r$ track back to a_j setting $a_i = r$ (speeds up subsequent processing)
Repeat for Q_2
Add the edge
if $r_1 \neq r_2$ **then**
  $c \leftarrow c - 1$ (two components are now joined)
else if $d_{r_1} < d_{r_2}$ **then**
  $a_{r_2} = r_1$
else if $d_{r_2} < d_{r_1}$ **then**
  $a_{r_1} = r_2$
else if $d_{r_1} = d_{r_2}$ **then**
  Q_1 and Q_2 are in the same component. A cycle has thus been found
end if
#cycles $= e' + c - v$ after e' relations have been processed

of a hash table T. However, this time we include a field to indicate the depth of each vertex. We use the roots of the components found by algorithm 5, which by default are chosen to be the smallest of the large primes in each component.

It is simple to identify odd and even cycles when using algorithm 6 - when we come across an edge whose vertices are both already stored in the graph, we consider the depth at which they are found. If they are at the same depth, then the cycle is odd. If they are at different depths, then the cycle is even. Note that depths will only ever differ by 1, since we are using a breadth-first traversal of the graph.

As shown in figure 4-3, assuming the bold edge is the last edge determining a cycle, we can compute the product of the edges marked a and divide by the product of the edges marked b to eliminate all large primes (assuming, of course, that the root vertex for the odd cycle is 1). Using such a procedure avoids the need to solve a (small) linear system to eliminate the large primes, and is thus rather more efficient. As we select the root of each component to be the smallest of the large primes acting as vertices in our

Algorithm 6 Build set of fundamental cycles of a graph

Hash component roots and add to the hash table at depth $D = 0$
WHILE unprocessed relations exist
$D \leftarrow D + 1$
LOOP through unprocessed relations
Get primes q_1 and q_2 for this relation
Hash to obtain indices i and j
Lookup
if Neither prime found in hash table **then**
  Defer
else if One prime (q_1 say) is found at index i **then**
  Add the other prime at index j:
  $d_j \leftarrow q_2$
  $a_j \leftarrow i$
  $depth_j \leftarrow$ current depth D
  Flag relation as processed
else if Both primes are found **then**
  Follow a_i pointers from both primes until they coincide
  Write edges in cycle to file
  Flag relation as processed
end if

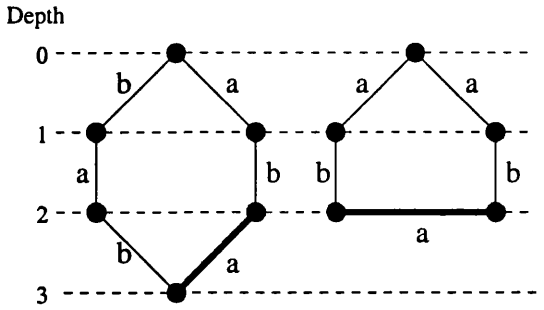

Figure 4-3: Ordering of edges for even and odd cycles

graph, we find that if there is any kind of 'choice' in the building of cycles (consider, for example, figure 4-4 where 6 cycles are possible but a fundamental set will only contain 3 cycles), the algorithm will generally build cycles passing through the root vertex. Thus were we to choose a different criterion to determine the root vertex, we would probably get a different breakdown in the number of odd and even cycles found (although the total would remain constant). If we add the special vertex 1 to the graph, this will always be the root vertex of at least one component, and the component containing the vertex 1 is usually the component containing *all* cycles in the graph. Note also that there is no guarantee that we shall get the shortest possible cycles in our result set – although this would be an advantage, as it would reduce the density of the full relations we may obtain from them. Strategies have been developed to try to reduce

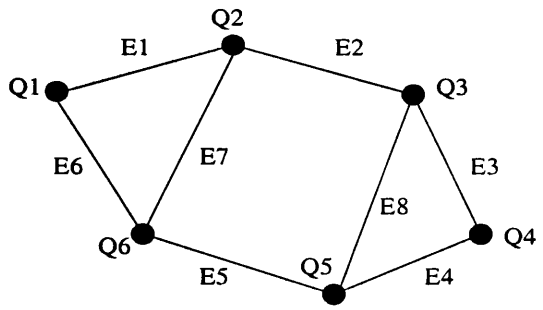

Figure 4-4: Simple graph with 3 fundamental cycles

cycle length from this procedure – see Denny and Müller [35] and Cavallar [20].

4.2 Maximising yield

We have shown that for discrete logarithm computation, since we need to divide out large primes, we have a slightly more complicated situation than occurs when the same method is applied to factoring. We now discuss how we can go about maximising the amount of full relations we can obtain from a given set of 1 and 2-partial relations.

4.2.1 Even more evens

Given a set of fundamental cycles for a graph, one would generally expect around half of these to have an even number of edges and around half to have an odd number of edges. As we can only benefit from using even length cycles in discrete logarithm computation – ignoring, for the moment, using the special vertex 1 – it would appear that the double large prime variant will not be as advantageous as it was found to be when applied to factoring. It is quite straightforward in principle, however, to obtain further useful cycles, even without adding the special vertex 1. Consider, for example, figure 4-4.

The cycle-finding procedures as described in algorithms 5 and 6 produce a set of fundamental cycles as desired. The majority of these cycles include the component root as one of their vertices. This happens to be the smallest of the large primes appearing as a vertex in that component. The graph in figure 4-4 will produce a different set of fundamental cycles depending on which vertex we specify as the root. If we assume vertex Q_1 is the root of this graph, algorithms 5 and 6 will produce cycles

- $E1 \rightarrow E7 \rightarrow E6$
- $E1 \rightarrow E2 \rightarrow E3 \rightarrow E4 \rightarrow E5 \rightarrow E6$
- $E1 \rightarrow E2 \rightarrow E8 \rightarrow E5 \rightarrow E6$

On the other hand, if we take vertex Q_6 to be the root, we find cycles

- $E6 \rightarrow E1 \rightarrow E7$
- $E7 \rightarrow E2 \rightarrow E8 \rightarrow E5$
- $E7 \rightarrow E2 \rightarrow E3 \rightarrow E4 \rightarrow E5$

In each case we have indeed found 3 fundamental cycles (as we would expect, since we have 1 component, 6 vertices and 8 edges). In each case we have 1 even cycle and 2 odd cycles. However, looking at the graph as a whole, we see that there are in fact 2 possible even cycles (of length 4 and 6) and 4 possible odd cycles. We can recover both of these even cycles *if* we can find two odd cycles having an edge, or even a vertex, in common. If we 'join' these two cycles together at their common edge or vertex, we can treat the composite shape as a new even cycle.

Consider the following situation. We have recovered a set of $2k$ fundamental cycles, and we assume for the sake of argument that we have k even length cycles and k odd length cycles in this set. We may keep the k even cycles and resolve as indicated above. We now represent the odd cycles as follows. Let each cycle correspond to a row in a matrix. The columns of the matrix map to the edges of the graph (or rather this component of the graph if it has multiple components). We enter a '1' in each column corresponding to an edge in a given cycle. We may thus treat each row of the matrix as a bitstring. If we compare any two rows with a bitwise AND operation, a non-null result string will indicate that the two cycles have at least one edge in common. We may now combine these two cycle with a bitwise exclusive OR operation to remove these common edges and create a new even cycle. From our k odd cycles we can thus create further even cycles. Care is of course needed in order to avoid linear dependencies among these new cycles. At best we would be able to create up to $k - 1$ independent even cycles, and process as for the k even cycles found directly. In such a situation – however unlikely it may seem – we would be only one cycle short of the number of practical cycles used in equivalent factoring methods where even and odd length cycles are equally useful.

These 'derived' even cycles have an obvious disadvantage in that they will generally be larger than the 'direct' even cycles which we built using algorithm 6. This has the knock-on effect of making the resulting full relations contain more coefficients, which in turn increases the density of the matrix we wish to solve. Furthermore, the relative size of the coefficients in these fulls is larger than those derived by other methods. We may thus suffer from coefficient blowup in some linear algebra routine that much sooner. It may be possible to follow the techniques of Denny and Müller [35], and try to reduce the length of the cycles that we have found. This technique uses similar ideas to those we have used to join cycles – once a set of cycles is found, we may look for those having a large number of edges in common and then 'subtract' one cycle from

another to form a new shorter cycle (although in a discrete logarithm setting, this may reintroduce the problem of unresolvable odd cycles). On the other hand, this increased density, if we could use it 'intelligently' in some way, should lead to more factor base elements being represented in at least one relation, and so may allow us to reduce the number of unknowns remaining after solution of the linear system. This would in turn speed up the final step of actual discrete logarithm computation - we shall discuss this matter further in later chapters.

4.2.2 1 as a 'large prime'

The implementation of Thomé [127] does not appear to distinguish between odd and even cycles, since partial relations are added to the graph and thus the vertex 1 is included as a 'large prime'. This reduces the need for even cycles as any cycle including 1 as a vertex may be resolved, be it of even or odd length. However, we may argue that the above method for building further even cycles from odd cycles is still valid for those cycles found which do not include 1 as a vertex. Again, due to the fact that many of the fundamental cycles built using algorithm 6 pass through the component root (taken to be the smallest vertex), we may expect that most of the cycles will indeed include 1 as a vertex (although some of these will be very long). The likelihood of finding odd cycles which possess an edge in common yet do not include 1 as a vertex - i.e. those we may join to form further even cycles - will thus be reduced, but we may still use the above procedure to gain additional full relations from our set of 2-partials. We also note that again we need to track the order of edges in a cycle, such that when dividing out the large primes, we are left with 1 on either the numerator or the denominator, and not one of the large primes Q_i.

4.2.3 Cycle possibilities

We can now examine the possible scenarios we may come across in practice, by considering figure 4-5. Recall that an edge Q_1Q_2 in the graph corresponds to a 2-partial relation of the form

$$Q_1Q_2 \prod q_i^{e_i}$$

where Q_1 and Q_2 are large primes and the q_i are elements of our pre-defined factor base. Resolving a number of such relations into a full relation means producing a relation

$$\prod q_i^{e_i}$$

composed purely of factor base elements. We note that the following possibilities may occur:

- Although rather uncommon, we can find relations where $Q_1 = Q_2$. Our 'cycle'

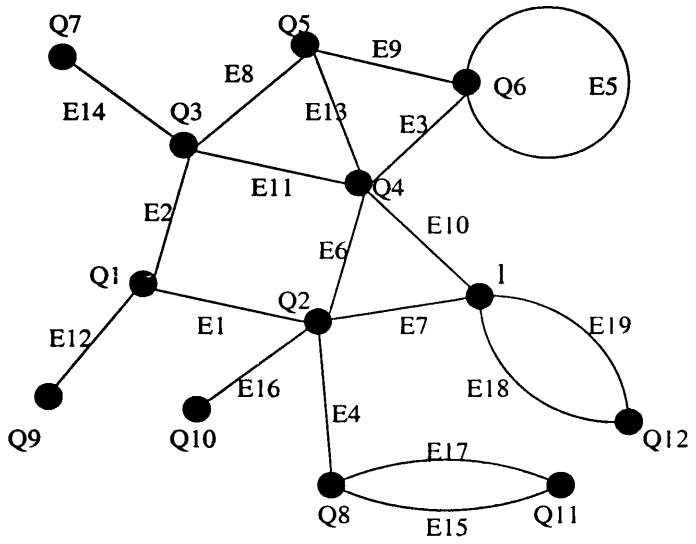

Figure 4-5: Graph illustrating possible cycles

is then of length 1, and is actually a loop in the graph such as E_5. Loops are simple to resolve for factoring, since the repeated large prime forms a square. For discrete logarithm computation, a loop may be resolved if we can find either two or more such loops (a still less likely occurrence), or alternatively if we can find one or more 1-partial relations, having large prime Q_1. We note that a loop at vertex 1 would correspond to a full relation. This latter situation will not occur.

- Rather more likely is that we have two or more relations involving the same two large primes, for example $E_{18} - E_{19}$ (corresponding to two matching 1-partials), or $E_{15} - E_{17}$. We then have cycles of length 2. These may be resolved by simply dividing the one relation by the other, and k matching relations would yield $k - 1$ independent full relations.

- We may find an odd cycle such as $E_6 \rightarrow E_7 \rightarrow E_{10}$, where one of the vertices is equal to 1. In this case, we may resolve the cycle by computing $(E_{10} * E_7)/E_6$. Again, note that we must be careful with the order in which we take the edges, since computing $(E_{10} * E_6)/E_7$ fails to eliminate Q_4.

- We may find an even cycle such as $E_6 \rightarrow E_1 \rightarrow E_2 \rightarrow E_{11}$. We can then resolve the cycle directly, irrespective of whether or not one of the vertices is 1, by computing, for example, $(E_{11} * E_1)/(E_6 * E_2)$

- We may find two odd cycles (such as $E_8 \rightarrow E_{11} \rightarrow E_{13}$ and $E_{13} \rightarrow E_9 \rightarrow E_3$) having an edge (as in this case) or simply a vertex in common. If they cannot be resolved in the above manner, we may resolve them by joining them together to

eliminate the common edge, and so form a new even cycle ($E_8 \rightarrow E_9 \rightarrow E_3 \rightarrow E_{11}$ for this example), which may be resolved as normal.

We now examine how different combinations of 1-partials and 2-partials affect the overall yield of the data from the index calculus procedure. To this end we will look briefly at 1-partials and 2-partials processed separately, and then examine the effect of combining all partial relations i.e. following the standard approach of treating 1 as a 'large prime'.

4.3 Results

Data was generated by using the basic index calculus method as described in the previous chapter[6]. Such data can be generated via any of the commonly used relation generation methods, from the basic 'trial and error' method used here to the highly sophisticated discrete logarithm Number Field Sieve. In a later chapter, we investigate a slightly different method of relation generation, which allows use of up to four large primes. Use of the basic method coupled with limited computing power, however, restricts us to looking at rather small examples. Discrete logarithms were computed modulo primes of up to 30 digits[7]. In each case, p was chosen such that $(p-1)/2$ is also prime. The generator g was not always chosen to be 2, nor necessarily a root of 2 such that we can remove the 2-column – g was simply taken to be the smallest generator. To give an idea of the relative yield of the basic index calculus method, consider figure 4-6. As one would expect, the number of relations grows linearly with the number of attempts made by the program. It is clear that the storage required to process partial relations is considerably more than that required if using full relations only. We can reduce the amount of partial relation data that we need to process by following Leyland et al. [81] and 'pruning' the datasets by recursively removing any relation which contains a large prime not occurring in any other relation (such relations are referred to as 'singletons'). All relations remaining after this pruning then form part of a cycle. Due to the amount of data we have to process, it may be best to break this into several processes having progressively better collision resolution, as described by Dodson and Lenstra [38].

4.3.1 Resolving 1-partials

As described previously, we resolve 1-partial relations via a hash table. A collision indicates two relations having the same large prime, and these can be divided to produce a full relation. As shown in table 4.1, we can obtain a considerable amount of full

[6]More sophisticated relation generation techniques were not employed, as the focus of the investigation was on the practicalities of resolving large prime variant data.

[7]See appendix B for parameters used.

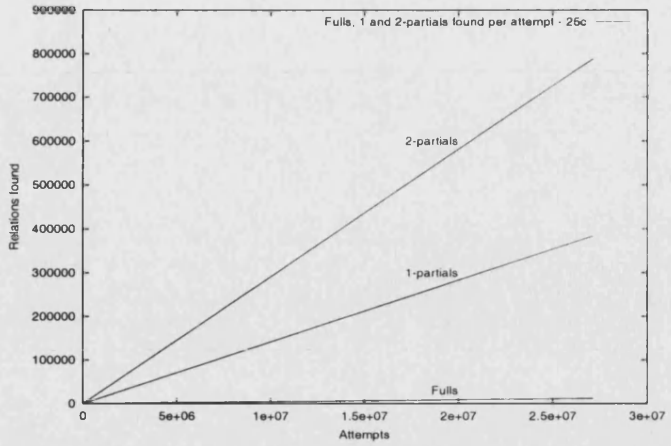

Figure 4-6: Distribution of relations – 25c

Dataset	Fulls (direct)	1-partials	Fulls resolved
20d	10,000	197,742	8,952
25c	12,000	383,844	13,765
30b	36,500	768,435	49,174

Table 4.1: Fulls resolved from 1-partials

relations from the 1-partials. Here we took $B_2 = {B_1}^2$ (slightly less than ${B_1}^2$ for dataset 30b), thus maximising the amount of relations obtained. Yield obviously increases at a greater rate as one takes more relations – see figure 4-7 for the amount of full relations we can obtain from a progressively larger amount of partial relations for dataset 30b. As shown by the log fit on the right, yield is roughly quadratic in the number of 1-partial relations processed.

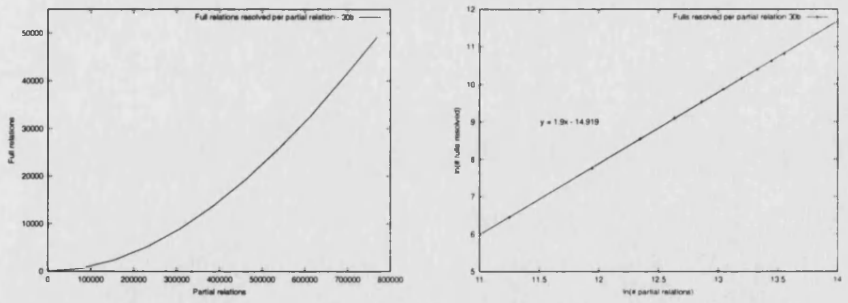

Figure 4-7: Number of fulls resolved per partial relation – 30b

An important point to note when using full relations derived from partial relations is that they are more dense – more factor base elements will generally be represented (in a full relation derived from partials) than would occur in a particular full relation obtained

'directly' in the relation generation process. When dividing one partial relation by another, some of the factors will coincide. In this case, the density of the subsequent full relation will either remain the same or be reduced, depending on the value of the exponents. The remaining factors will add to the density of the resolved full relation. We note the increases in density for the three datasets in table 4.2. It can be seen that the density of fulls derived from 1-partial relations is roughly 1.5 times the density of fulls obtained directly; as a certain proportion of the elements are common to both relations. A result from Knuth [66, section 4.5.2] states that for two integers chosen at random, the probability that they are relatively prime is $\frac{6}{\pi^2}$. There is obviously a $(\frac{1}{2})^2$ chance that two relations both contain the factor 2, a $(\frac{1}{3})^2$ chance that they share the factor 3, and so on. In fact, if we compute

$$S = \prod_{p \leq B} (1 - \frac{1}{p^2})$$

for the values of B used here, we find that S matches $\frac{6}{\pi^2}$ to 5 decimal places, so it is likely that two 1-partial relations will share factors with probability $1 - \frac{6}{\pi^2} \approx 0.39$. The additional criterion that all but one of the prime factors of our two numbers are below some bound B, and that the other factor is always shared (the common large prime Q), does not appear to affect this estimate in practice. It is reasonable then to assume that we will often have a certain amount of overlap.

Dataset	Nonzeros per full (direct)	Max value	Nonzeros per full via 1-partials	Max value
20d	7.7	24	10.9	18
25c	8.9	17	13.3	21
30b	9.7	17	14.9	23

Table 4.2: Density of fulls resolved from 1-partials

We also see that, in addition to the average density of the full relations increasing by some 50%, the maximum coefficient value may also increase somewhat. It turns out that, in each of the above cases, this maximum value corresponds to the exponent for factor base element 2. One would of course expect 2, as the smallest element in the factor base (ignoring the possible inclusion of -1, whose exponent value is always either 0 or 1), to have the largest exponent, on average, in a given full relation. This underlines the advantage of having either 2, or some root of 2, generate the finite field $(\mathbb{Z}/p\mathbb{Z})^*$, so we may ignore these values in practice.

Estimating yield

One can deduce the number of full relations that can be obtained from a set of 1-partials if one knows the number of unique large primes and the number of relations R in the dataset. In general, the first part of this information will not be readily accessible, so it would be useful to be able to compute some estimation of yield based on R and the large prime bounds used. An approximation for this is given by Morain [91], but here we again follow Lenstra and Manasse [75], who propose using the formula

$$R - \#Q + \sum_{q \in Q} (1 - P_q)^R$$

where Q is the set of primes q with $B_1 < q \leq B_2$ and P_q is the probability that a large prime q occurs in a particular relation. This last is approximated by

$$P_q \approx q^{-\alpha} / \sum_{p \in Q} p^{-\alpha}$$

Here, $\alpha < 1$ is some positive constant to be determined by experiment. Using this approximation, we obtain the estimates in table 4.3.

Dataset	1-partials	α	Fulls resolved (estimated)	Fulls resolved (actual)
20d	50,000	0.76	676	698
20d	100,000	0.76	2,508	2,523
20d	150,000	0.76	5,254	5,346
25c	100,000	0.74	1,047	1,048
25c	200,000	0.74	3,908	4,077
25c	300,000	0.74	8,233	8,766
30b	250,000	0.71	6,132	6,002
30b	500,000	0.71	22,421	22,466
30b	750,000	0.71	46,205	47,077

Table 4.3: Estimated and actual yield of 1-partials

Lenstra and Manasse note that, for their implementation, taking $\alpha \in [\frac{2}{3}, \frac{3}{4}]$ gave a reasonable estimate. Denny (according to Boender and te Riele [16]) takes $\alpha = 0.775$, suggesting that such a value is implementation-dependent. Following [16], we computed α for our implementation by using Maple to estimate yield for $\alpha = 0.70, 0.71, \ldots, 0.80$ after gathering a certain number of 1-partials for each dataset. Taking the value giving the best fit gave the values of α noted in table 4.3. Carrying out such an analysis is quite straightforward, and gives a useful indication of the progress of relation generation.

4.3.2 Resolving 2-partials

To resolve 2-partial relations, we use graph-theoretic techniques to build cycles among the 2-partial relations, and thus eliminate the large primes to produce further full relations. In this section we consider only 'pure' 2-partials – we do not allow 1 as a large prime. We are thus restricted to looking for cycles of even length, or possibly joining odd cycles with a common edge or vertex, as mentioned previously. We will refer to this latter case as a 'derived' even cycle. Table 4.4 shows the number of cycles found for the three datasets. We first filtered out any 'duplicate' 2-partial relations (those having the same large primes Q_1 and Q_2) and resolved them as we did the 1-partials above. Thus all cycles in table 4.4 have ≥ 3 edges.

Dataset	2-partials	Fundamental cycles	Even cycles	Odd cycles	Derived even cycles
20d	238,018	241	112	119	118
25c	788,237	7,009	3,480	3,529	3,524
30b	1,494,109	57,946	28,775	29,171	29,168

Table 4.4: Fulls resolved from 2-partials

In addition to these cycles, we can add the full relations gathered via 1-partials (which are, of course, cycles of length 2). We have not included loops in table 4.4 – there were none for dataset 20d, 3 for dataset 25c and 2 for dataset 30b. A loop is a relation of the form

$$P_1 P_1 \prod p_i^{e_i}$$

We can eliminate the large primes in such a relation by either finding a second loop and dividing the two relations, or by finding a partial relation

$$P_1 \prod q_i^{e_i}$$

and dividing the 'loop' relation by the square of this partial relation. Processing the above loops in this manner, we – unsurprisingly – found no duplicate loops, but did manage to resolve 4 full relations for dataset 25c, and 2 further fulls for dataset 30b – so while it is true that one can make use of these relations, yield is of course negligible. The number of fulls we can resolve for the 30 digit dataset by using 1 and 2-partial relations is in fact greater than the total number of fulls we obtained directly. The restriction of needing even cycles is offset by the amount of even cycles we can derive by joining odd cycles – it seems surprising that one can find so many odd cycles with a common edge. However, this is a by-product of the manner in which we have built a set of fundamental cycles. We find that in each of the examples above, all cycles appear in the same component, and most of them pass through the component root: thus

increasing the chance of finding a common edge. We note that one could in fact join cycles having simply a vertex in common. Given the success of joining cycles having an edge in common, this was not attempted except to resolve loops. In these examples, there were no cycles of length 2 (i.e. duplicates) among the 'pure' 2-partials. We can see how yield increases (as we take more 2-partials in our graph) in figure 4-8 which refers to dataset 30b. Yield is roughly quartic in the number of 2-partial relations processed.

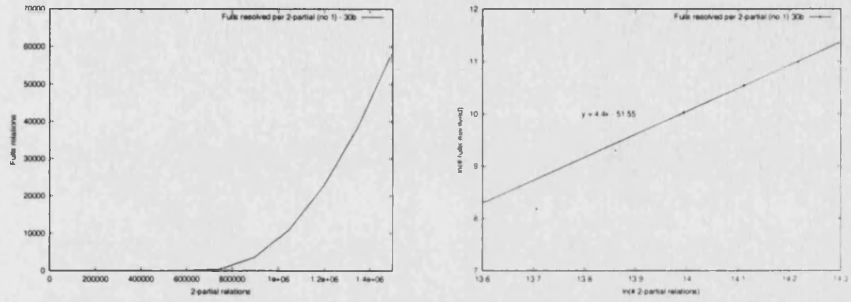

Figure 4-8: Number of fulls resolved per 2-partial relation without 1 as a vertex – 30b

It is interesting to note the distribution and average length of these cycles, as it obviously has a bearing on the density of the full relations which we can resolve from them. There are 1,494,109 2-partial relations in the graph for dataset 30b. Our depth-first search tells us that these are split across 171,478 components. However, due to the method of connecting components during this cycle counting procedure (algorithm 5), we find that one component is considerably larger than all the others. This component is the one with the smallest root, which for the 30 digit dataset happens to be 300,007 (the smoothness bound for this particular dataset was 300,000). This component contains 1,162,291 relations – the next largest component contains only 68. As one may expect, given this distribution of relations, all 57,946 cycles found in this graph occur in this large component (all other components are trees); and in fact 43,569 of these cycles include the prime 300,007 as a vertex. As it happens, the prime 300,007 actually occurs in 25 of the 1,494,109 2-partial relations in this dataset. It is reasonable, then, to expect that the cycles will generally have a length ≥ 5 (odds) and ≥ 6 (evens), and this is indeed the case. The average even cycle has some 26 edges, while the average odd cycle also has around 26 edges. Joining odds together to form further even cycles results in cycles having an average of 35 edges.

As before, we can examine the density of these full relations. Consider table 4.5.

We see that full relations obtained via 2-partials are far more dense than the fulls we obtained directly, and also considerably more dense than the fulls we have obtained via 1-partial relations. Further, the maximum coefficient found in the sets of full relations found from the 2-partial relations is also quite a lot larger. As noted, there are two points to raise concerning the question of density. On the one hand, the more dense

Dataset	Nonzeros per full via even	Max value	Nonzeros per full via derived even	Max value
20d	122.8	34	152.7	35
25c	99.98	43	125.06	53
30b	107.5	41	139.5	41

Table 4.5: Density of fulls resolved from 2-partials

the matrix, the harder it will be to solve, as our only real advantage was its sparse structure. On the other hand, one could argue that we will represent more elements of the factor base if we use these denser relations, thus helping us when we later come to discrete logarithm computation. These points will be investigated in chapters 6 and 7.

4.3.3 Resolving 1 and 2-partials together

We can, of course, remove the restriction of requiring even cycles by adding the vertex 1 to the graph. We may then resolve odd cycles into full relations if they include this special vertex. Since most cycles built by the method of Lenstra and Manasse [75] pass through the root vertex, we may hope that the majority of the odd cycles we find will indeed include 1 as a vertex and as a result may be resolved directly: with only a few odd cycles not including 1 as a vertex being candidates for the cycle joining procedure carried out as before. In the tables that follow, we have filtered out any duplicate relations i.e. those having the same two large primes. These are then resolved separately as in the previous section.

Dataset	2-partials	Fundamental cycles	Even cycles	Odd cycles	Derived even cycles
20d	426,808	12,778	4,102	8,676	0
25c	1,158,316	39,658	15,232	24,426	0
30b	2,213,370	185,151	73,502	111,649	0

Table 4.6: Fulls resolved from 2-partials and 1-partials combined

We see firstly that we have increased our yield considerably. The 1-partials may still be resolved amongst themselves, so we need only compare results in tables 4.4 and 4.6. Again we have not included loops; which occur in the same amounts as before (since adding 1-partials to the graph will not create any more or less of these).
All odd cycles (with the exception of these loops) included 1 as a vertex and thus could be resolved directly. There was no need to join odd cycles to form further even cycles, apart from the loops, where yield has of course remained the same. Yield for the 20 digit dataset has thus increased from 240 fulls from 2-partial relations alone to 12,778 fulls if we combine them with the 1-partials.

We can compare the yield we get when adding 1 to the graph to that which we obtained when processing 2-partials alone – see figure 4-9. Yield when processing with 1 is rather less than cubic for this example, but we obtain cycles with far fewer relations thanks to the frequently occurring vertex 1.

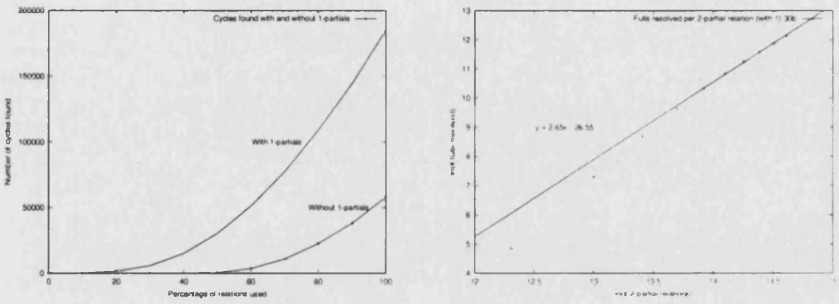

Figure 4-9: Cycles found with and without 1 as a vertex – 30b

As before, all cycles found for dataset 30b occurred in the largest component, which has root 1. However, whereas before the root vertex of this component occurred in only 25 relations, we now have that the root vertex occurs in 719,261 relations – it occurs in every 1-partial relation (here we ignore duplicates). As a result, we get shorter cycles – for dataset 30b the average even cycle had 4.5 edges and the average odd cycle had 3.9 edges. This clearly reflects in the relative densities of the full relations these cycles yield: see table 4.7. Had we been required to join odd cycles, as we were previously, a derived even cycle would contain $a + b - 2n$ edges if the two odd cycles had a and b edges respectively and had n edges in common, which will lead to denser relations than those found by using odd cycles with 1 as a vertex since $a, b \geq 3$ for all cycles considered here.

Dataset	Nonzeros per full via even	Max value	Nonzeros per full via odd	Max value
20d	17.25	19	15.08	21
25c	23.07	25	20.5	29
30b	26.79	24	23.92	34

Table 4.7: Density of fulls resolved from 2-partials and 1-partials combined

As the smallest even cycle we find (ignoring pairs, or cycles of length 2) is of length 4, while the smallest odd cycle (ignoring loops) is of length 3, we find that the full relations resolved from odd cycles are slightly less dense, on average. Maximum coefficient size is slightly larger for the fulls resolved from odd cycles. Going back to figure 4-5, we resolved an odd cycle including 1 as a vertex by computing $(E_{10} * E_7)/E_6$. This 'imbalance' of having one more edge in the numerator than the denominator may result

in our slightly larger maximum exponent.

Density and coverage

The only advantage of increased density is that we can represent (or 'cover') more factor base elements in a given set of full relations, and hopefully solve for a larger proportion of the factor base. This will increase the efficiency of the final step of the index calculus method, that of actually computing discrete logarithms.

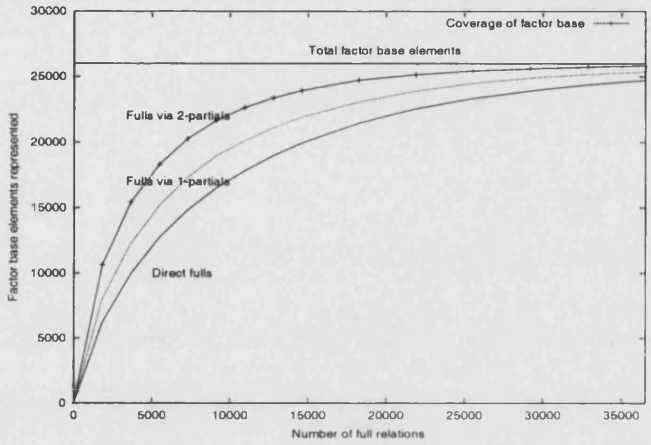

Figure 4-10: Number of factor base elements represented by full relations – 30b

Figure 4-10 shows 4 different plots for dataset 30b. The constant line is the number of elements in our original factor base. The curves represent the number of these elements represented by full relations, full relations resolved from 1-partial relations, and fulls resolved from 2-partials. We see that we may take slightly fewer fulls resolved from partials – in particular those resolved from 2-partials – yet maintain the same number of unknowns represented by these relations. This may allow us to take fewer excess rows in our linear algebra routine, which we do both to maximise the number of factor base elements represented in our matrix, and to try to ensure that we obtain a solution – however, the increased density of the matrix may slow processing down to an extent that outweighs any subsequent benefits.

4.3.4 Overall speedup

We now look at the effect of using partial relations on the time we spend building relations. If we firstly ignore the effort needed to process partial relations – which is, of course, not insignificant – we may try to estimate the point at which we may terminate relation generation.

Effect on relation generation time

It would be useful to try to estimate the point at which our relation generation code would have produced enough full relations (either directly or via the use of partials) to build a matrix with a given number of rows. We can try to put a value to this by working backwards and comparing the yield if we were only to process some percentage of the 1-partials and 2-partials found overall. In this way we can build up a picture of the 'true' yield of the procedure, showing how many fulls we can expect to resolve in total at a given stage in the procedure. We may then see how the quoted speed up factor of 2-2.5 from factoring given by Lenstra and Manasse [75] compares to the same procedure as applied to discrete logarithm computation.

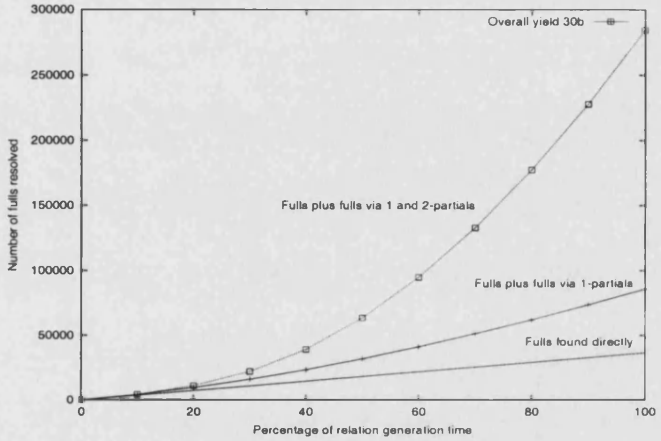

Figure 4-11: Overall yield – 30b

Figure 4-11 shows the amount of fulls we obtain directly for the 30 digit dataset compared firstly to the amount of fulls we get when we include the fulls we can resolve using 1-partials, and then to the amount we can obtain when also including those we get from resolving 2-partials (taking the maximum yield, i.e. including 1 in the graph as discussed previously). If we look at the point at which, by resolving 1 and 2-partial relations, we get as many fulls in total as we can get directly, we can see that the speedup factor is indeed around 2–2.5 times for discrete logarithm computation using 1-partials and 2-partials; in that we may achieve our final total of 36,500 fulls in less than 50% of the time taken to build a similar number of full relations. In fact, the speedup factor gets greater as we go from the small, 20 digit dataset to the larger 30 digit dataset, as one can note by considering figure 4-12. Note that the term 'speedup' here is a little deceptive – we are considering purely the time spent in relation generation, and disregarding the amount of effort we need to expend, and indeed the amount of storage we need, in order to actually build and resolve the 1-partials and 2-partials. This effort is not inconsiderable, particularly in terms of storage. Lenstra and Manasse

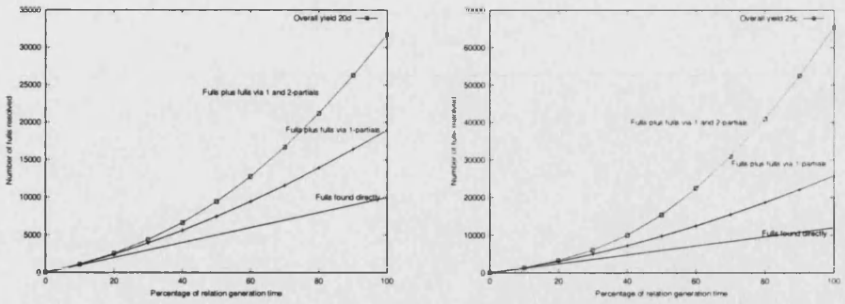

Figure 4-12: Overall yield – 20d and 25c

[75] reduce the large prime bounds somewhat in an effort to reduce the amount of data to process (particularly the number of 2-partials). In this study we have set our 2-partial bound B_3 to be ${B_1}^3$, that is the cube of the original factor base bound (actually, slightly smaller for dataset 30b). While it was our intention, in setting these bounds, to obtain larger dataset sizes, one of course cannot compare directly between datasets for computation of 30 digit discrete logarithms and 'cutting edge' datasets gathered in computing discrete logarithms of over 100 digits. Were one to use similar bounds to ours at this level, there would be a vast amount of data to process, and we would have to look at parallelising or otherwise improving the performance of the resolution of these relations.

Reducing dataset sizes

The benefits we can obtain by using partial relations will certainly become hampered by the amount of such relations which we can reasonably expect to store and process. This is particularly relevant to the 2-partial relations. Choosing the smoothness bound B_3 to be rather less than ${B_1}^3$ should *not* in fact have a commensurate effect on the yield we can expect from the graph processing; in that it is unlikely that a given number – our remainder after smoothness testing – is a product of 2 primes both close to our large prime limit B_2. As a result, by lowering the bound B_3, we lose only those relations which were unlikely to contribute to cycles in a significant manner.

Consider figure 4-13. This shows the total number of cycles we can find, had the smoothness bound B_3 for dataset 30b been a lower percentage of its actual value. If we reduce B_3 by 50%, the number of cycles we can resolve into further full relations drops by only around 3.5%. Cutting B_3 by 95%, our yield drops from some 180,000 cycles to around 140,000 cycles; a drop of only some 23%. Had our bound been 0.5% of its actual value, we would still have found 73,767 full relations via 2-partials; which is actually twice what we obtained direct from the relation generation phase of the index calculus method. Further, in this case we would reduce the number of 2-partial relations in our dataset (before pruning) from 1,494,019 to 165,876 – a reduction of

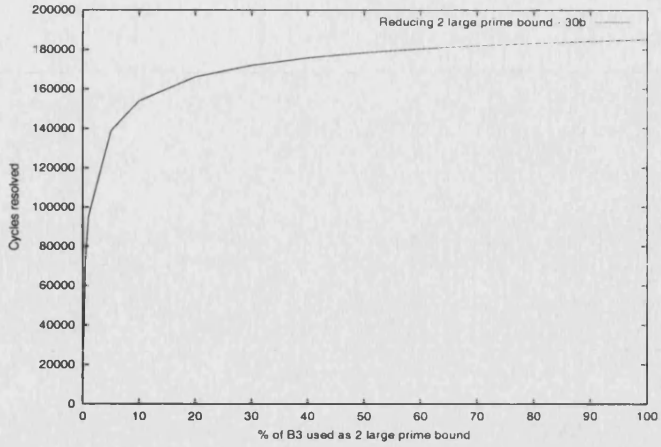

Figure 4-13: Effect of reducing 2-prime smoothness bound – 30b

89%. Since each relation consists of 2 large primes, one 30 digit value and roughly 6 further values (the other factors of $g^a \bmod p$), this represents a considerable saving in space and in processing time for resolving 2-partials; and more importantly would have increased the speed of original relation generation due to a reduced number of primality tests and Pollard ρ computations.

Cost of resolving partials

We now consider the effort required to process partial relations. Firstly, of course, there is an additional load placed on whatever relation generation method we are using. When storing 1-partials only, this is merely a check to see if the remainder is less than our large prime bound B_2. When using 2-partials, however, we add the cost of a primality test and an application of Pollard ρ (or some other factoring method) in order to identify the large primes. As shown in table 4.8, this has quite a serious effect on performance. Here we show the time taken to store 500 full relations, and subsequently the number of 1-partials and 2-partials we can obtain alongside these 500 fulls, with the time taken to detect and store these partial relations. Notice that the time increase to generate 2-partials is actually getting less pronounced as we go to larger dataset sizes.

Dataset	Fulls	Time	+1-partials	Time	+2-partials	Time
20d	500	5,211	9,767	5,224	11,611	45,983
25c	500	76,262	16,211	76,361	31,891	247,766
30b	500	903,893	10,719	907,665	51,465	1,511,876

Table 4.8: Timings for generation of 500 fulls with and without storing of partials

We subsequently have to resolve these partial relations. Firstly, it makes sense to strip any useless relations from the datasets. These 'singletons' contain primes which do not occur in any other relation. Following Dodson and Lenstra [38] we stored the hash of the large primes without collision resolution, and discarded any which only occurred once. This procedure was repeated until the datasets were reduced to a sufficient level to repeat this procedure *with* collision resolution. Timings are detailed in table 4.9 – here 'passes' refers to the number of pruning passes through the datasets.

Dataset	1+2-partials in	1+2-partials out	Time to prune	Passes taken
20d	435,760	47,425	1,967	8
25c	1,172,083	123,006	6,240	9
30b	2,262,544	510,097	19,512	9

Table 4.9: Timings for pruning of datasets

To resolve the 1-partials, we simply need to match up large primes. Timings – operating on 'pruned' data – are shown in table 4.10. Resolving 1-partials actually takes very little effort even if the datasets have not been pruned. Pruning 1-partials alone for dataset 30b took 3,022 hundredths of a second, and it then took 343 hundredths of a second to resolve the remaining relations into fulls. For comparison, if we had *not* pruned this dataset, it would have taken us 1,861 hundredths of a second to resolve the 49,174 full relations.

Dataset	1-partials	Pruned	Fulls	Time to resolve
20d	197,742	16,105	8,952	48
25c	383,844	24,925	13,765	86
30b	768,435	87,821	49,174	343

Table 4.10: Timings for resolving 1-partials

When resolving 2-partials (here we consider maximising yield, i.e. adding 1 to the graph and processing 1 and 2-partials together), we have a rather more involved procedure. Determining the graph structure (algorithm 5) takes a number of operations slightly more than linear in e, the number of edges in the graph (Lenstra and Manasse [75]). Building cycles via algorithm 6 depends on how many passes through our set of e relations we make – in the worst case, we would scan the dataset e times, but in practice we only require $k < e$ passes, where for our implementations[8] k was at most 8. In practice, then, runtime is $O(e^{1+\epsilon})$. We also have the additional task of multiplying out the relations to eliminate the large primes. Timings are shown in table 4.11.

[8]Here k did increase slightly (from 6 for dataset 20d to 8 for dataset 30b) as larger datasets were considered.

Dataset	(Pruned) 1 + 2-partials	Fulls	Time to count	Time to build	Time to resolve	Total
20d	47,425	12,778	57	90	71	218
25c	123,006	39,658	170	326	250	746
30b	510,097	185,151	768	1,496	1,287	3,551

Table 4.11: Timings for resolving 1 and 2-partials via graph approach

Going back to table 4.8, we note that generating 12,778 fulls for dataset 20d would take 133,172 hundredths of a second. Resolving this many relations from partials took just over 2000 hundredths of a second, including time taken to prune the datasets. Obviously, this isn't the whole story, as the time taken to generate the partial data particularly the 2-partials – means that a better evaluation of the effectiveness of using 1 and 2-partials comes from timing at the point in figures 4-12 and 4-11 where we could have terminated relation generation. For dataset 20d, for example, we generated 10,000 full relations. Had we stopped after obtaining 5,000 – i.e. after 50% of the time we spent – we could have recovered almost 5,000 further fulls from the 1 and 2-partials. For dataset 25c this point came after around 40% of the time actually spent, and for dataset 30b it came after some 35%. Considering timings at these points, we obtain table 4.12.

Dataset	Fulls	Time	Fulls via 1+2-partials	Time	Total time
20d	5,000	459,830	4,435	796	460,626
20d	9,435	98,331	•	•	98,331
25c	4,800	2,378,553	5,232	2,079	2,380,632
25c	10,032	1,530,120	•	•	1,530,120
30b	12,775	38,628,431	16,713	4,658	38,633,089
30b	29,488	53,307,993	•	•	53,307,993

Table 4.12: Timings for direct fulls compared to using 1 and 2-partials

Here, the second line for each dataset shows the time taken to build a certain amount of full relations directly – i.e. *not* storing any partial relations. The top line shows the time it would have taken to obtain this many relations, had we stored and resolved 1 and 2-partial relations. We see that, for dataset 20d, the additional effort required to store 2-partial relations far outweighs their usefulness. For dataset 25c, it is still better to simply use full relations directly. For dataset 30b, however, we now save time by terminating relation generation, and relying on our partial relations to make up the shortfall; suggesting that the double large prime variant does indeed give a practical speedup for larger modulus sizes. For this particular example, we do not have the 2-2.5 times speedup quoted by Lenstra and Manasse [75] – we can obtain this in terms of

yield, as shown by figures 4-12 and 4-11, but not in terms of time. However, the fact that we have shown that we can resolve all but a fraction of relations which would also be useful for factoring suggests that we will obtain this speedup for larger dataset sizes.

Dataset	Fulls	Time	Fulls via 1-partials	Time	Total time
20d	6,000	62,688	3,452	366	63,054
20d	9,452	98,508	•	•	98,508
25c	6,000	916,332	5,291	783	917,115
25c	11,291	1,722,148	•	•	1,722,148
30b	18,250	33,129,772	13,712	1,471	33,131,243
30b	31,962	57,780,456	•	•	57,780,456

Table 4.13: Timings for direct fulls compared to using 1-partials

Carrying out similar timings for using 1-partials only, we obtain table 4.13. Due to the negligible extra work needed to obtain 1-partials during the relation generation step, it is *always* worth using the single large prime variant. The crossover point to going from one large prime to two for factoring is generally accepted to be for numbers of around 70 digits Lenstra and Manasse [75]. For smaller numbers, the two large prime variant is less effective. These other results use more efficient methods of relation generation than we have used here; but we have seen a similar effect at smaller scales, in that for our implementation, we find that the basic index calculus method using two large primes does give practical speedup (compared to using at most one large prime) for computation of discrete logarithms modulo a prime of 30 or more digits.

4.4 Summary

In this chapter we have investigated the practical differences between the application of large prime variants (with up to two large primes) to discrete logarithm computation, compared to their well-documented application in factoring.

The key difference between the use of these techniques for factoring and for discrete logarithm computation is the lack of restriction on the type of cycle found when using a graph theoretic approach to resolving 2-partials for factoring. For the discrete logarithm case we are working in a large finite ring rather than modulo 2, and so must divide out the large primes. We have demonstrated how this can be done as cycles are built, by a careful consideration of the nature of the cycles and the ordering of their edges. This allows us to avoid more complex methods of eliminating the large primes, such as solving a linear system. Adding the special vertex 1 to the graph allows us to process 1 and 2-partials together. This then allows greater yield and shorter cycles leading to sparser full relations, as in factoring. For discrete logarithms, however, it

has the additional important benefit of allowing us to resolve odd cycles so long as they include this special vertex. By considering all possible cycles in the graph, one can try to maximise the yield from a given dataset. We have shown that it is possible in some cases to join odd cycles together to form further even cycles, and to combine loops with 1-partials. The effectiveness of these last techniques is however very small when compared to the improvements obtained by simply adding 1 to the graph, and the density of the relations we obtain may be against their use in the traditional linear algebra step; but they are nonetheless valid and may be of use in some way during a back-substitution phase. Weber [132] reports that, in certain cases, some 2.8% of cycles found in an implementation for discrete logarithm computation could not be resolved. Our experiments are much smaller, but using the above techniques allowed us to recover a full relation for all but ε cycles in the graph. The few exceptions consisted of one or two isolated loops which could not be matched to a 1-partial, and in fact ε was generally zero.

Use of partial relations brings many practical considerations. Datasets can become huge, and simply storing and processing this data can become awkward, although various pruning strategies can be invoked to ease this load. The price of the success of processing 1-partials and 2-partials together is the increased processing and memory requirements of the graph algorithm, not to mention the increased effort to generate such data in the first place. However, we have shown that it is actually possible to *resolve* partial relations for discrete logarithm computation such that yield is within some ε of that which we would obtain if solving modulo 2 in a factoring application. Our experiments suggest that the quoted speedup of 2-2.5 obtained in factoring will carry across directly to discrete logarithm computation.

Chapter 5

Towards n Large Primes

In this chapter we build on the work of the previous chapter by examining the impact of using partial relations with more than two large primes. In order to generate such data, we make use of the so-called *Waterloo variant* of the index calculus method. This allows us to use a 'double double' large prime variant, and we subsequently investigate the practicalities of resolving n-partials for discrete logarithm computation.

5.1 More large primes

Given the performance improvements to relation generation brought about by use of the single and double large prime variants of the index calculus method, it is natural to consider going further and using three or more large primes. Indeed, for factoring purposes this has been attempted, with promising results – Leyland et al. [81] use up to three large primes per relation in a Quadratic Sieve implementation, while Cavallar [21] uses up to three large primes per smoothness test in a Number Field Sieve implementation. One difficulty now, however, is in bounding the large primes one can obtain – it is straightforward to define another bound B_4 as the fourth power of our factor base bound B_1. We may then trial divide using our factor base, and check if our remainder is less than this new bound B_4 (but greater than B_3). The problem is that now we do not have so simple a situation as in the double large prime case. Then, we could say with certainty that a composite remainder was the product of exactly two large primes, both greater than B_1 but less than B_2. Stepping up to 3 large primes, we lose a certain amount of this control, since we know that we will have either two or three factors in our remainder, but we cannot bound these factors as closely as before. Allowing three large primes, we cannot tell whether a composite remainder is indeed the product of three large primes, or the product of two large primes where one of these is larger than bound B_2. If we solely concern ourselves with relations producing large primes within our bounds B_1 and B_2, then we will potentially obtain a large number of useless relations where one large prime is greater than B_2. Since actually obtaining this

large prime factorisation involves one application of Pollard ρ (in the 'false report' case) or two applications (three large prime case), it would seem that these extra relations would be too costly to bother with. However, results of [81] indicate that, whilst there is indeed added complexity in both obtaining these three large prime relations and in resolving them into useful full relations, it is in fact worth the effort and may lead to an important improvement in algorithm performance when factoring larger numbers.
Dodson and Lenstra [38] describe an implementation of the number field sieve using four large primes. This is a rather different approach to that used by Leyland et al. [81], since the four large primes of [38] consist of two lots of two large primes. This is done by applying the double large prime variant to each of the NFS factor bases – both the algebraic factor base and the rational factor base. This leads to *two* sets of large primes (or rather large prime ideals), which generally cannot be resolved against each other[1]; so this can be thought of as a 'double double large prime' variant rather than a 'four large prime' variant in the same vein as the three large prime technique described above. A similar approach was taken by Weber [132] to compute discrete logarithms, although few details are given concerning the resolution of partial relations.
We now consider how the techniques of the previous chapter scale to the use of more large primes by examining a similar 'double double large prime' technique; again concentrating on how we can actually *resolve* the partial relations we obtain, and on identifying how such methods differ from the factoring application. However, here we generate data such that we have a *single* set of large primes, in an attempt to examine how the techniques of Leyland et al. [81] adapt to the discrete logarithm situation.

5.2 The Waterloo variant

A simple improvement to the basic index calculus method is offered by the so-called 'Waterloo variant' of the index calculus method, outlined by Blake et al. [13, 14] and Coppersmith [26]. This is another method aimed at improving the efficiency of the relation generation step. The original paper describes the method as applied to computations over GF(2^n), but it is equally applicable over GF(p).

5.2.1 Overview

Rather than testing some value $A = g^a \bmod p$ for smoothness, as we did in the basic method, we now use the extended Euclidean algorithm (algorithm 7), to find values u_1, u_2 and u_3 such that

$$u_3 = u_1 A + u_2 p$$

[1]When using an NFS-type approach with two factor bases allowing one large prime on each side, we note that the graph of 2-partial relations is then bipartite. Thus *every* cycle is even, and so the techniques of the previous chapter should allow all cycles to be resolved directly.

where both u_1 and u_3 are smooth. Then computing

$$A = g^a \equiv u_3 {u_1}^{-1} \bmod p$$

leads to a full relation. The description in algorithm 7 comes from Knuth [66]. Note that $u_1 a + u_2 b = u_3$ throughout the algorithm.

Algorithm 7 Extended Euclidean algorithm

Input: Integers (a, b)
Output: (u_1, u_2, u_3) such that $u_1 a + u_2 b = u_3 = \gcd(a, b)$
  Initialise:
  $(u_1, u_2, u_3) \leftarrow (1, 0, u)$
  $(v_1, v_2, v_3) \leftarrow (0, 1, v)$
  Process:
  while $v_3 \neq 0$ **do**
    $q = \lfloor u_3 / v_3 \rfloor$
    $(t_1, t_2, t_3) \leftarrow (u_1, u_2, u_3) - q(v_1, v_2, v_3)$
    $(u_1, u_2, u_3) \leftarrow (v_1, v_2, v_3)$
    $(v_1, v_2, v_3) \leftarrow (t_1, t_2, t_3)$
  end while

Of course, on termination of the extended Euclidean algorithm on input A and p, u_3 will be 1 since p is prime; however, the absolute sizes of *both* u_3 and u_1 are smallest when $u_1 \approx \sqrt{p}$, and at this point we exit extended Euclid. The idea is that if u_1 and u_3 are both $O(\sqrt{p})$ then there is a better chance that they are both smooth than there is simply of $A = g^a \bmod p$ being smooth. Previously, we noted that the chance of a number $< p$ being B-smooth was approximated by the value u^{-u} as $u \to \infty$, where

$$u = \frac{\log p}{\log B}$$

If we instead consider the chance of finding two smooth numbers, both $O(\sqrt{p})$, we have a probability of $(\bar{u}^{-\bar{u}})^2$ where now

$$\bar{u} = \frac{\log \sqrt{p}}{\log B}$$

which thus gives us a probability of approximately $(\frac{1}{2}u)^{-u}$ This does not change the asymptotic performance of the algorithm, but does improve the $o(1)$ value in the complexity estimate, giving practical savings. Again we can use partial relations, this time arising from two separate factorisations – that of u_1 and u_3. The downside of this approach is that we now require two smoothness tests and two applications of Pollard ρ (or whatever method we are using to factorise the remainder). Thomé [127] used a variation of this technique to compute discrete logarithms modulo 2^{607}. He allowed at most two large primes in a full relation – either both from u_3 or one from both u_1 and

u_3.

Use of the Waterloo variant means that we must 'merge' the two smooth values u_3 and (the inverse of) u_1 to obtain a full relation. However, we note that these values do not contain any shared factors.

Lemma 5.2.1. *On input (a, p) for prime p, the extended Euclidean algorithm computes values (d_i, x_i, y_i) such that $d_i = ax_i + py_i$ and $\gcd(x_i, d_i) = 1$.*

Proof. Clearly $\gcd(a, p) = 1$ since p is prime. We also have that $\gcd(x_i, y_i) = 1$ by induction on i. Suppose now that $\gcd(x_i, d_i) \neq 1$, so there is some value $z_i > 1$ which divides both x_i and y_i. Then z_i must also divide py_i, and, since p is prime, z_i must divide y_i. However, this means that z_i is a common factor of both x_i and y_i, and contradicts the fact that $\gcd(x_i, y_i) = 1$. We conclude that $\gcd(x_i, d_i) = 1$. □

We do not then have overlapping factors reducing the density of the ensuing full relations (as we did when merging two 1-partial relations, for example). As a result, the number of nonzeros in a full relation is simply $\omega(u_1) + \omega(u_3)$ where $\omega(u_i)$ is the number of distinct prime factors of u_i. We give an illustration of the Waterloo procedure in figure 5-1, which shows the number of 'full' relations obtained with each iteration of Extended Euclid (averaged over the number of attempts needed to generate 500 full relations for a 40 digit p). Here, for illustration purposes, we have started to test u_1

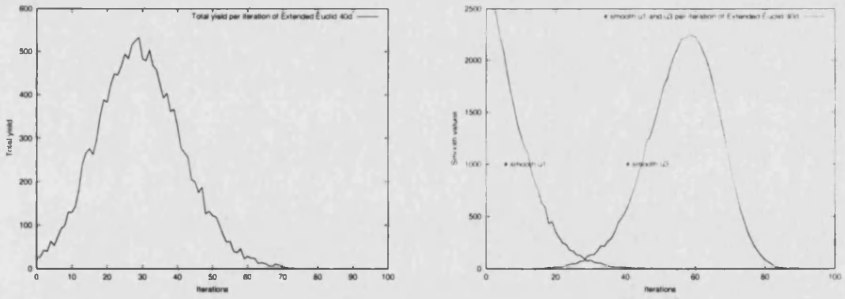

Figure 5-1: Smoothness testing in Extended Euclid – 40d

and u_3 for smoothness when u_1 is less than $C\sqrt{p}$, where for this example $C = 10^{15}$ and $\sqrt{p} = 10^{20}$. In the left hand figure, we show the total number of relations (fulls and 1 – 4-partials) found for successive iterations of extended Euclid. We see that this is at a maximum after some 30 iterations – and this is in fact the point at which u_1 is approximately $\sqrt{p}$. In the right hand figure, we see the reason for this higher yield – it is at this point that we find *both* u_1 and u_3 are most likely to be smooth.

5.2.2 Yield of Waterloo variant index calculus

The purpose of the Waterloo variant is to improve the chances of finding smooth values leading to 'full' relations. We would thus expect to take fewer attempts overall to find

a particular number of full relations than we would by using the basic procedure. Consequently, we would also expect to find fewer partial relations involving 1 or 2 large primes than we did previously, due to testing fewer values. To give a brief illustration of this, table 5.1 shows the yield of the Waterloo variant (WV) procedure compared to the basic index calculus (IC) method, when using identical parameters to generate relations modulo a 40 digit prime. The time taken to generate the data will be discussed later in this chapter. Again, details concerning parameters used in the generation of the datasets used in this chapter are given in appendix B.

Method	Attempts	Fulls	1-partials	2-partials	3-partials	4-partials
IC	2.5×10^6	13	819	3,247	-	-
WV	2.5×10^6	424	90,772	526,408	500,191	127,639

Table 5.1: Basic index calculus v Waterloo variant - 40 digit p

It is clear that the Waterloo variant is extremely beneficial in practice compared to the basic method. Relations generated via the Waterloo method had, in general, one more factor than their counterparts generated by the basic method (the same goes for respective 1 and 2-partial relations).
In this chapter we will examine how use of the Waterloo variant technique affects our use of large prime relations.

5.3 Using large primes

Using the Waterloo variant coupled with the methods of the previous chapter, we can use up to four large primes in the relations we generate (two in both u_1 and u_3). An advantage of using two lots of two large primes with the Waterloo approach, rather than than using more than three large primes in the standard manner (as per Leyland et al. [81]), is that the large primes are constrained – using two sets of two large primes allows us to force all these values to be within certain bounds, which should cut down on the list of false reports. However, using more large primes, no matter how the relations are obtained, obviously calls for more processing in order to resolve useful relations from them, as we now discuss.

5.3.1 Waterloo with 1-partials

Using large primes in the Waterloo variant procedure means that we must now consider the exponent of the large primes, since considering 1-partials we may obtain relations of the form

$$g^a \equiv Q_1 \prod q_i^{e_i} \bmod p$$

$$g^a \equiv \frac{1}{Q_1} \prod q_i^{e_i} \bmod p$$

As one would expect, there is a 50% chance of getting either type. We may resolve these much as before – the only change concerns the exponent of the large prime in each of the relations being considered, since this governs whether or not one should multiply or divide in order to eliminate the large prime.

5.3.2 Waterloo with 2-partials

Resolving 2-partials from the basic relation generation method came down to the fact that odd cycles required one of the vertices to be the special vertex 1 in order to eliminate all large primes, while even cycles could be resolved without this additional criterion. Care is needed in order to process cycle edges in the correct order such that all the large primes cancel.

Examining 2-partial relations obtained via the Waterloo variant procedure, we find that we may now have relations of the form

$$g^a \equiv Q_1 Q_2 \prod q_i^{e_i} \bmod p$$

$$g^a \equiv \frac{1}{Q_1 Q_2} \prod q_i^{e_i} \bmod p$$

$$g^a \equiv \frac{Q_1}{Q_2} \prod q_i^{e_i} \bmod p$$

$$g^a \equiv \frac{Q_2}{Q_1} \prod q_i^{e_i} \bmod p$$

Obviously, the first two types are equivalent to all intents and purposes, as are the second two types; so effectively we have two different kinds of relation to consider. However, combinations of the two types do affect the outcome of the graph resolve procedure described earlier. Where before we were looking for even cycles, and could only resolve odds if one of their vertices was equal to 1, we must now take account of 'matching exponents' – basically, whether we have an even or odd number of the second type of relation. If we have an even number of these, then we have the same outcome as before – an even cycle can be resolved directly; an odd cycle requires 1 as a vertex. If on the other hand we have an odd number of the second type of relation, the situation is reversed, as shown in figure 5-2.

Consider firstly the two odd cycles on the left. The first is the same as that encountered in the previous chapter – all primes have exponent 1. In order to resolve this, we found that we needed one of the large primes Q_i to be the special vertex 1. Use of the Waterloo variant, however, allows cycles such as the second of these two odd cycles. Here we have a 'mismatch' in the exponents – edge E_2 now links Q_3 to $1/Q_2$. No matter how we invert any of the edges in this cycles, we will always have a mismatch

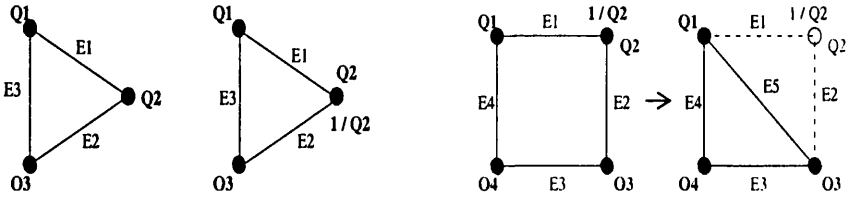

Figure 5-2: Effect of exponents (left) and reducing cycles (right)

like this. Now, however, we no longer need the special vertex 1 to be present. We can simply compute

$$\frac{E_1 E_2}{E_3}$$

to eliminate all the large primes. We are effectively reducing a cycle to the situation we had previously, as shown in the right hand side of figure 5-2. We may multiply two relations together to eliminate one of the large primes, which in effect creates a new edge E_5, where $E_5 = E_1 E_2$, such that E_5 is of the form

$$E_5 = Q_1 Q_3 \prod q_i^{e_i} \bmod p$$

This then reduces the number of edges in the cycle by 1, removes the exponent mismatch, and allows us to resolve a full relation as we did previously i.e. an even cycle can be resolved directly whereas an odd cycle requires 1 as a vertex.

One must thus keep a still closer eye on the exact nature of cycles, and only a strict adherence to a particular edge order will allow a cycle to be resolved into a full relation. As it happens, in all datasets used in this study, some 10% of the 2-partial relations were of the form $Q_1 Q_2$, while around 90% were of the form $\frac{Q_1}{Q_2}$ – it is more likely that we will have u_1 and u_3 both being 1-partials than it is that, for example, u_1 is smooth and u_3 is a 2-partial. This does not dramatically change the cycle counting and building procedures described in the previous chapter – it is only in resolving these cycles that additional care is needed. In practice, when one considers adding an edge (i.e. relation) to the graph, one should invert[2] the new edge if necessary such that the exponents of the vertices (i.e. the large primes) are the same. When an edge is found to have both vertices already in the graph, we will have a cycle which can be resolved as before if and only if the exponents of these vertices match. If, however, one of the vertices has a different exponent while the other matches, we must multiply the two relations where these large primes with differing exponents occur.

[2]By 'invert' we simply mean to change the sign of all exponents in the relation – since we hold all these values this is not a problem.

5.3.3 Waterloo with 3, 4, and more-partials

We have the following possibilities when considering 3-partial relations obtained via the Waterloo variant procedure, both occurring with equal probability

$$g^a \equiv \frac{Q_1 Q_2}{Q_3} \prod q_i^{e_i} \bmod p$$

$$g^a \equiv \frac{Q_1}{Q_2 Q_3} \prod q_i^{e_i} \bmod p$$

Since by inverting one of these we find the other, we may consider these to be the same. 4-partial relations only arrive in the form

$$g^a \equiv \frac{Q_1 Q_2}{Q_3 Q_4} \prod q_i^{e_i} \bmod p$$

Processing 3 and 4-partial relations would obviously require extensions to the graph algorithms. As before, we can trim our datasets by recursively removing any relation (be it 1, 2, 3 or 4-partial) which contains a large prime which does not occur in any other relation. The effect of this pruning on a 40 digit dataset is illustrated by table 5.2.

Dataset 40E	1-partials	2-partials	3-partials	4-partials
All data	58,962	799,818	3,508,419	5,144,673
Pruned	20,172	110,091	263,721	277,302

Table 5.2: Pruning partial relations – 40 digit p

The amount of data needing to be fed into our extended graph algorithms is now very much reduced, and we have the advantage of knowing that all 'edges' in our graph (or, rather, our hypergraph) do indeed form part of a cycle (or, rather, 'hypercycle') of some kind.

Basic n-partial resolve

Visualising exactly what these cycles look like and how to resolve them is rather more awkward than it was when we processed 1 and 2-partials alone. To illustrate the situation we now face, and the kind of approach we could take, consider figure 5-3. Here we process a 3-partial relation (or indeed, a relation with n large primes) with the 1 and 2-partial relations already processed. To guard against building cycles which are linearly dependent on those we have already built when resolving the 1 and 2-partials, we remove the last edge from each cycle found and output a tree of 1 and 2-partial relations. We assume that the root of this tree is the vertex 1.

We now process 3-partial relations one at a time. We do not at present add them to

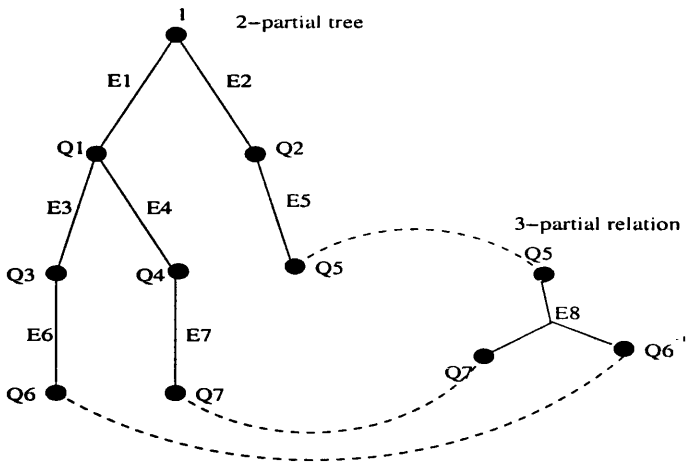

Figure 5-3: Resolving 3-partials – basic case

the graph, we simply try to find a relation having all as three primes in the 2-partial tree. In figure 5-3, we have found that the 3-partial E_8 has all three vertices in the tree. We may now track back to the root from each vertex. In each of these 'trails', all large primes will appear an even number of times, with the exception of the root vertex; but as this is 1 it is not important. For the example given, then, we may compute

$$t_1 = \frac{E_6 E_1}{E_3} \quad t_2 = \frac{E_4}{E_7 E_1} \quad t_3 = \frac{E_2}{E_5}$$

Note that we have taken account of the exponents of the vertices of E_8. Then computing $E_8 * t_1 * t_2 * t_3$ gives (ignoring the factor base elements and noting only the large primes in the relations)

$$\frac{Q_5 Q_7}{Q_6} * \frac{Q_6 Q_3 Q_1 1}{Q_3 Q_1} * \frac{Q_4 Q_1}{Q_7 Q_4 Q_1 1} * \frac{Q_2 1}{Q_5 Q_2} = 1$$

We then obtain a relation involving only factor base elements. This basic resolving procedure easily extends to 4-partials and indeed n-partial relations. Further, we can improve on it slightly in certain situations – notice in the example above that trails t_1 and t_2 meet at vertex Q_1. We may then terminate the two trails *if* the exponent of Q_1 as it first appears in trail t_1 is different to that of Q_1 as it first appears in trail t_2. This would reduce the length of the two trails (assuming the intersection is at a vertex other than the root vertex), and would thus lead to a lower density in the final full relation. The above illustration does not give us a simple way of processing more than one 3 or 4-partial at a time. The need to divide out the large primes makes the use of n large prime variants more complex for discrete logarithm computation than it is when applied to factoring methods, where we simply look for all primes occurring an even number of times. Extending the algorithms of the previous chapter is difficult, since

if processing 3-partials and actually adding them to the hypergraph, we can have the situation where we find all three vertices, but do not in fact have a *complete* hypercycle. Further, it is easy to find examples of hypercycles which *cannot* be resolved – consider figure 5-4. Here we have a 3-partial with repeated large prime Q_1, together with a

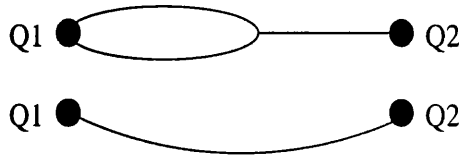

Figure 5-4: Simple yet unresolvable hypercycle

2-partial (we ignore exponents). The two relations form a simple hypercycle, yet we cannot eliminate all large primes – for factoring or for discrete log – since Q_1 occurs an odd number of times (assuming that these are 'genuine' partial relations and that the vertex 1 is not present).

General n-partial resolve

We now discuss how may go about resolving relations involving more than two large primes. It would be nice to recover the situation of being able to assume that we have found a hypercycle *if* we have found all vertices of a relation under consideration already present in the graph (or hypergraph). We can do this to a certain extent by relying rather more on our dataset pruning. If this is done correctly, we know that *all* remaining relations form part of some hypercycle. We can then build our hypergraph as we built the graph before; with the only real change being that we must process n vertices for a given n-partial. If we find all n vertices are already stored in the hypergraph, then we could store this relation in a 'proto-cycle' file. Once we have processed all relations, we may then turn our attention to these proto-cycles, tracking through the original (pruned) dataset, and building hypercycles as we match vertices. This strategy works up to a point – one can isolate a subset of relations which together form a hypercycle. However, it is difficult to efficiently track through the hypergraph due to the 'branching' introduced by using relations having more than two large primes. Where previously the method for building cycles used the concept of depth in the graph, and cycles were found by tracking back to the root, we may now be forced to track 'down' the hypergraph as well as 'upwards' towards the smallest vertex. We are also hampered by the need to 'divide out' our large primes, which again is not such a problem for factoring variations of this technique.

The trick of ordering edges in the cycle no longer applies, and again it is not necessarily true that we can obtain the same yield as in an analogue factoring application. Consider the three relations

$$\{E_1, E_2, E_3\} = \{Q_1 \prod {q_{1_i}}^{e_{1_i}}, Q_2 Q_3 \prod {q_{2_i}}^{e_{2_i}}, Q_1 Q_2 Q_3 \prod {q_{3_i}}^{e_{3_i}}\}$$

where the q_j are factor base elements as before. Here, all large primes occur twice (except for 'large prime' 1), and thus together they form a hypercycle, as shown in figure 5-5. This hypercycle differs from the other example in that we *can* resolve it by

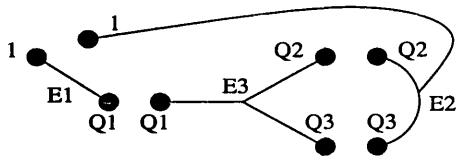

Figure 5-5: Simple hypercycle which can be resolved for factoring

dividing the 3-partial by the product of the 1-partial and the 2-partial. This, however, assumes that all exponents are 1. For Waterloo-derived data, either one or two of Q_1, Q_2, Q_3 will have exponent -1. If we consider the possibilities for the exponents of the primes, we find that, even had we built it, we only have a 50% chance of resolving this hypercycle. We are no longer rescued by the presence of 1. For factoring purposes, however, solving modulo 2 would always allow us to resolve such a hypercycle[3].
A further annoyance is that it is no longer so simple to count the number of hypercycles in the hypergraph. Leyland [80], working with up to three large primes per relation, proposes generalising the basic formula for fundamental cycles which is used by Lenstra and Manasse [75]. Rather than using the formula

$$\#\text{cycles} = E + C - V$$

for E edges, C components and V vertices, he considers both edges and 'arcs'; where an arc is defined as a link between any two vertices. Thus a 2-partial relation can be thought of as a single edge, and a single arc. A 3-partial, however, is still a single edge but creates 3 arcs in the hypergraph. Going further, a k-partial will create $\frac{k(k-1)}{2}$ arcs. The basic formula can then be thought of as

$$\#\text{hypercycles} = A + C - V - 2E$$

for a hypergraph of 3-partials. As noted by Leyland, however, such a generalisation does not hold, and can only give an approximation. By adding the vertex 1 to some relations, we create loops in the hypergraph which are not hypercycles, yet get counted as such. If we do not count those arcs which are in fact loops, we can get a negative

[3]This example is actually one noted by Leyland [80] as a hypercycle of a kind not picked up by his cycle resolving algorithm. As it turns out, it is also a good example to highlight the difficulties of the discrete logarithm three large prime technique as opposed to its application in factoring.

hypercycle count. If we do not add 1 to relations having fewer than n large primes, we lose uniformity in the hypergraph. Although counting the cycles has no particular bearing on actually resolving partial relations into useful full relations, it does make it more difficult to estimate yield when considering when the relation generation stage is complete.

A rather simpler method of resolving partials is that offered by Dodson and Lenstra [38]. As it turns out, this is essentially the same method as that used by Weber [131, 134] and by Leyland et al. [80, 81], as we now discuss. The method attempts to reduce n-partial relations into $(n-1)$-partial relations by means of 1-partials. As a result, all cycles and hypercycles obtained include at least one 1-partial. The cycle-finding methods of Lenstra and Manasse [75], which we used to build cycles among 1 and 2-partials, created cycles which – generally – all involved the special vertex 1, so in fact all cycles resolved here also included at least one 1-partial relation (unless 2-partials were processed alone, which lead to denser full relations anyway). The method relies in part on the dataset being pruned. This gives us the set of all relations occurring in cycles. It also allows us to determine the number of vertices V and edges E in the graph (or rather hypergraph), from which we can estimate the number of 'hypercycles' present by computing $E - V$, on the assumption that 'pruned' edges form a single component.

Having pruned the dataset, one considers the remaining 1-partials. They are firstly resolved among themselves, and then all unique large primes amongst this set of relations are removed from the 2-partials, 3-partials and 4-partials. Remaining relations are then resorted into 1, 2, 3 and 4-partials (or, rather, into either 1-partials or 'more than 1'-partials), and the process is repeated until no further relations are either resolved or reduced. Any remaining relations – hopefully few in number – may then be resolved by some kind of graph resolve as before. Weber uses this strategy on 3 and 4-partials, and subsequently uses the standard graph resolve on the ensuing set of 1 and 2-partials. Leyland et al. use a similar method to build chains of relations, which can subsequently be multiplied up to eliminate the large primes (modulo 2). However, for discrete logarithm purposes, it is not so straightforward to eliminate the large primes in such a chain. We note, though, that hypercycles such as those in figures 5-4 and 5-5 would *not* be built by such a procedure. The first example does not contain a 1-partial, and in order to build the second example, we need another 1-partial with large prime Q_3 or Q_4.

The key point when using 1-partials to reduce higher-order partial relations is that we *always* find an even number of edges incident at any vertex in the hypercycle, with the exception of the special vertex 1. Thus, for factoring purposes one can eliminate the large primes modulo 2 simply by multiplying all edges in the hypercycle. We can link back to our earlier simple illustration of resolving 3-partials one by one by considering

the hypercycles found by such a procedure as trees, as shown in figure 5-6. It can now be seen that we can divide out all large primes in the hypercycle as we did before, by following 'branches' back to the special vertex 1.

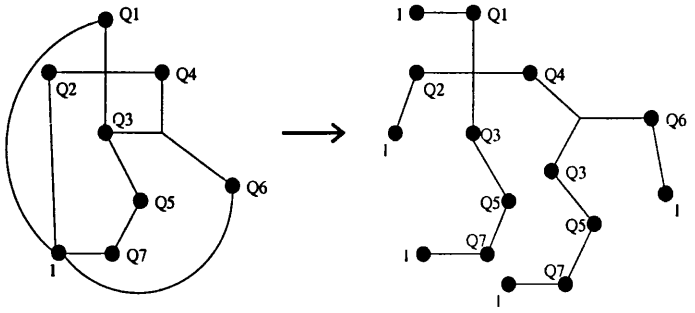

Figure 5-6: Hypercycle as a 'tree'

The actual hypercycle is shown on the left. What is built by recursively removing 1-partials is the 'tree' on the right. Now every vertex (except maybe the vertex 1) occurs in an even number of edges, and tracking round the tree one can eliminate all the large primes. Again, we must take account of the order in which we process the edges – if we do not maintain the order of edges as built, we are in danger of hitting a problem at a vertex such as Q_3 in the example, in that we may have a choice of direction to take. If we follow the order in which the hypercycle was built, we will always take a 'depth first' route through the tree and end up at a 1-partial.

Since this method of building hypercycles means that the vertex 1 is always present at the end of every branch, we do not have the unresolvable cases of odd cycles without the vertex 1 (including loops) which we had previously. We can thus resolve *all* hypercycles that are actually built by this technique, both for factoring and for discrete logarithm computation. We did find that, as noted by Dodson and Lenstra [38], after the build procedure, occasionally one is left with a small subset of partials (having more than one large prime) which together form a cycle. In practice, for our implementations, these remaining relations were always 2-partials. Some of these were loops, which could not be resolved (since no 1-partials match the repeated prime, and unsurprisingly we did not find two loops at the same prime). The others were pairs of 2-partials, having the same primes Q_1 and Q_2. As these are even cycles, these could be resolved. Any odd cycles among the remaining edges would probably cause problems, since 1 is not present in the graph component under consideration. If relations remain, then, after recursively removing 1-partials, we may not quite reach the same yield as in factoring. Another point to note about figure 5-6 is that, in the case of processing 1 and 2-partials only, the methods of Dodson and Lenstra [38] and Leyland et al. [81] will often create slightly longer cycles than the method of Lenstra and Manasse [75] – the repeated path from Q_3 to the special vertex 1 is unnecessary, as we could have stopped at vertex Q_3

where the paths join. We sometimes have the extreme case where we actually resolve a loop using two duplicate trails, as shown in figure 5-7. Again, the hypercycle is shown on the left, with what we can actually build on the right. Note that for factoring, again, loops are trivially resolvable.

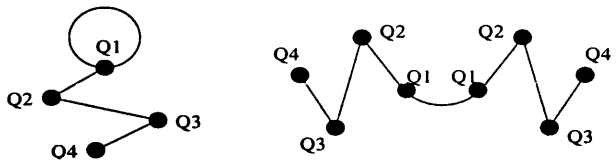

Figure 5-7: Resolving loop via duplicate paths

For higher order partial relations, we do need to follow paths all the way to vertex 1 in order to guarantee removal of all large primes. This suggests that, if using 1 and 2-partials only, the method of Lenstra and Manasse [75] may be preferable. The length of cycles can be reduced further by a strategy such as that described by Denny and Müller [35], or via the linear algebra techniques of Cavallar [20].

Maintaining edge order

In order to build a set of hypercycles, we followed the method of Leyland et al. [81]. To briefly summarise the implementation, one separates partials into 1-partials and '>1'-partials, as mentioned. After resolving the 1-partials, one removes the set of unique large primes they contain from the '>1'-partials, adding the relevant 1-partial to a list, or 'chain' associated with each relation. The process is then repeated until no further changes are made. As 1-partials are resolved, both the relations themselves *and* their chains are output as hypercycles. To facilitate processing, Leyland [80] uses two hash tables for this procedure – one contains lists of primes keyed by relation number, and the other lists relations in which each large prime occurs, keyed by the hash of this large prime.

For factoring purposes, one can simply multiply all relations in each hypercycle to eliminate the large primes modulo 2. In order to divide out large primes for discrete logarithm purposes, we used the following procedure. We firstly store the first relation in the hypercycle chain as a pseudo-full relation – this will initially have between one and four large primes, which we wish to eliminate. We make use of a hash table to store relations keyed by each large prime they contain. For each large prime in the chain, therefore, we have a list of relations in which it occurs. Since each relation in the chain can only be used once, we flag a relation inactive once it has been processed. We then eliminate the primes in our pseudo-full relation one by one by looking up an active relation involving this prime and dividing or multiplying accordingly.

The key part of this procedure turns out to be in the choice of relation with which to

eliminate a particular prime. We look up the relation list for this prime, but will often have some sort of choice to make. We must take care to choose the first active relation which immediately followed the last relation processed in the chain. In this way, we follow the ordering with which the chain was built, and will then always end up at a 1-partial.

Of course, eliminating a large prime will often *add* to the number of large primes in our pseudo-full relation. We must keep a certain order to these large primes - if we are resolving prime Q_1 from the pseudo-full which at this point has primes $Q_1Q_2Q_3$, using a relation having large primes $Q_1Q_4Q_5$, then in order to take a depth first route through the hypercycle, it is convenient to store the primes in the revised pseudo-full as $Q_4Q_5Q_2Q_3$. In this way, we keep track of any forks in the path and maintain a strict ordering of edges.

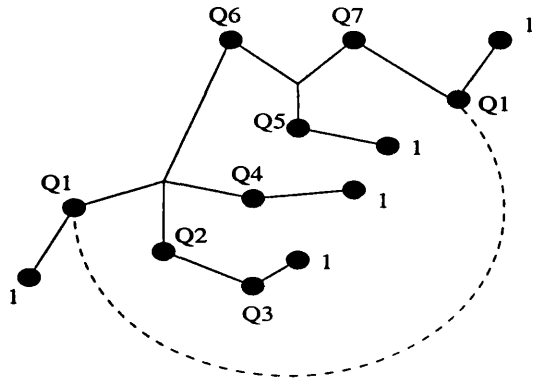

Figure 5-8: Hypercycle involving 1, 2, 3 and 4-partials

To illustrate the importance of maintaining edge order, consider figure 5-8 which shows a typical hypercycle involving a 4-partial, a 3-partial, two 2-partials and five 1-partials. The vertex 1 is repeated in order to clarify the diagram, and to emphasise its special character as a 'universal destination'. Notice that vertex Q_1 occurs twice. At some point we will consider prime Q_1, and look to eliminate it. We have a choice – do we use one of the two 1-partials, or do we use the 2-partial Q_7Q_1? Assume we have eliminated all primes but Q_1 and Q_7. If we use the 2-partial to eliminate Q_1, then we cannot use it again at a later point. There is subsequently no relation involving prime Q_7 when we come to try to resolve it – it will be stored twice in the pseudo-full relation. It may well be the case that it is stored as $Q_7^1Q_7^{-1}$, i.e. we could actually cancel it; but the point is that the procedure outlined above will fail to resolve the cycle. It is again important, then, to follow the order of edges carefully – by doing this we can *always* resolve the cycles that are built.

5.4 Results

In this section we detail implementation results using the Waterloo variants coupled with the use of up to four large primes. Processing for 1 and 2-partials, as discussed, is much the same as it was previously and results are noted rather briefly, before we examine the impact of using 3 and 4-partials.

5.4.1 Resolving 1-partials revisited

As noted, resolving 1-partial relations is not essentially any different to that described previously for the basic index calculus method. Yield for dataset 40d is shown in figure 5-9. As before, we see that the number of full relations resolved grows with the square

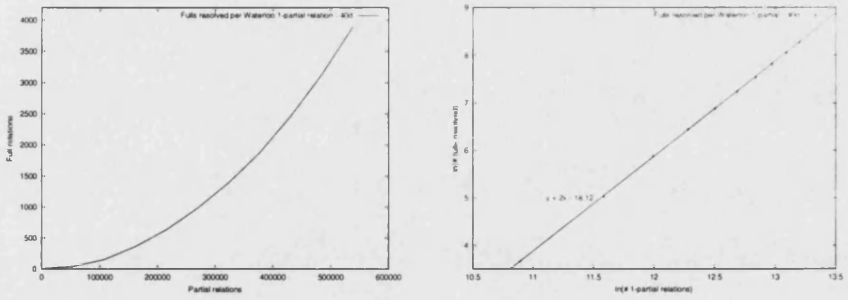

Figure 5-9: Yield of 1-partial relations – 40d

of the number of 1-partial relations processed. Resolved relations had on average some 20 nonzero elements, with a maximum coefficient size of 23, compared to direct fulls having 13 nonzeros on average. This is in keeping with previous results, since 1-partial relations themselves had, on average, 11 nonzero factor base elements (plus, of course, one larger prime).

Estimating yield

In the previous chapter, we used the estimate of Lenstra and Manasse [75] to predict yield for a given number of 1-partial relations. It is interesting to note that this estimate, for the examples considered here, does *not* give the same results when considering different methods of relation generation. In table 5.3 we show the number of fulls we could obtain from a particular number of 1-partial relations, when generating datasets using either the basic index calculus method (ICS) or the Waterloo variant (WV). Identical parameters were used in each case.

Yield when processing relations obtained via the Waterloo variant is greater than that obtained via the basic method. This is due to the nature of the Waterloo procedure. The single large prime in each 1-partial comes from a smoothness test carried out on

Dataset	1-partials processed	Fulls resolved (ICS data)	Fulls resolved (WV data)
20d	15,000	68	291
20d	30,000	260	1,103
20d	45,000	572	3,311
25c	30,000	94	384
25c	60,000	387	1,502
25c	90,000	865	3,234
30b	100,000	1,011	2,265
30b	200,000	3,908	8,345
30b	300,000	8,511	17,783

Table 5.3: Yield of 1-partials – basic index calculus versus Waterloo variant

a number of size $O(\sqrt{p})$ rather than on a number of size $O(p)$, as was the case in the previous chapter. Since the number that we are testing is much smaller, it is more likely to have smaller factors. This may well create a larger number of smaller large primes (and associated grammatical chaos). Figure 5-10 plots the relative sizes of the large primes in the first 1,000 1-partials found by each technique. While the difference is not hugely pronounced, it can be seen that for the Waterloo data on the right there is rather greater clustering at the lower end of the large prime range. This encourages further matches, and we obtain more full relations.

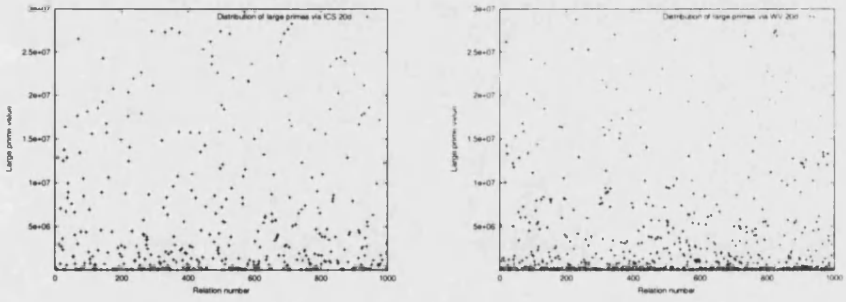

Figure 5-10: Distribution of large prime values – 20d

Going back to estimating yield, the probability that a particular large prime q occurs in a given 1-partial is assumed to be

$$P_q \approx q^{-\alpha} / \sum_{q \in Q} q^{-\alpha}$$

Lenstra and Manasse [75] originally took $\alpha = 1$, but found that the resulting estimates were consistently too high. The reason for introducing α is to take account of the fact that we are using smooth numbers. Our candidate numbers turned out to be

smooth but for this large prime factor q. Since smaller numbers are more likely to be smooth, this may make the appearance of larger large primes more likely, and so α is used to take account of this possible bias. In chapter 4 we found $\alpha = 0.76$ to give a good prediction of yield for dataset 20d. Now, however, by using the Waterloo variant, we may reduce the effect of this bias by increasing the occurrence of smaller primes. This should lead to larger values of α being required to give good estimation of yield. Computing these new values, we obtain table 5.4.

Dataset	1-partials	α	Fulls resolved (estimated)	Fulls resolved (actual)
20d	15,000	0.91	290	291
20d	30,000	0.91	1,084	1,103
20d	45,000	0.91	2,281	2,354
25c	30,000	0.87	424	384
25c	60,000	0.87	1,593	1,502
25c	90,000	0.87	3,378	3,234
30b	100,000	0.82	2,357	2,265
30b	200,000	0.82	8,723	8,345
30b	300,000	0.82	18,224	17,783

Table 5.4: Estimated and actual yield of 1-partials via Waterloo

We see that we do indeed need to take a larger value for α when using the Waterloo variant. We also see that the value for α is falling as we take large values of p – it may well be that for larger dataset sizes we get back to $\alpha \in [\frac{2}{3}, \frac{3}{4}]$ as observed by Lenstra and Manasse [75]. This would require further testing.

5.4.2 Resolving 2-partials revisited

The main practical difficulty in resolving 2-partials is in tracking through the graph whilst maintaining a strict ordering on the edges of a cycle – other than this, processing is much the same as it was previously. Yield – roughly cubic – is detailed in figure 5-11.

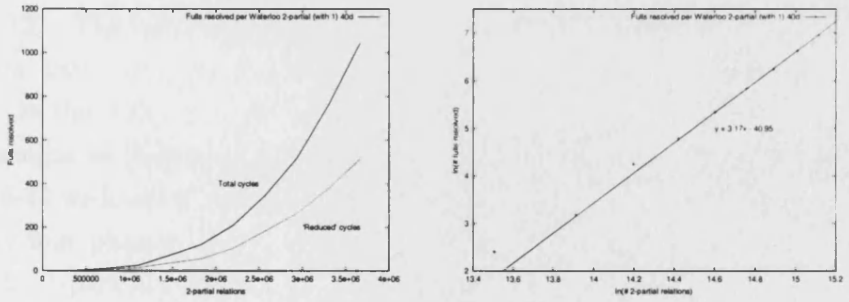

Figure 5-11: Yield of 2-partial relations – 40d

In the left-hand plot, the dotted line indicates how many cycles had an 'odd parity' and were reduced via edge multiplication to a cycle with exactly one less edge. This reduced cycle could then be resolved as normal. As before, the downside of using partials is an increase in density – the average resolved relation has some 29 nonzero entries, while the average 2-partial had around 9 (plus two large primes).

5.4.3 Resolving 3 and 4-partials

We now consider the effect of 3 and 4-partials. We suffer slightly in that we are only able to investigate small examples if we use the basic index calculus method – for the examples here, we are computing modulo a 40 digit prime. This means that we need to take quite a small smoothness bound in order to actually see much benefit from using the 3 and 4-partials – consider figure 5-12. Here $B_1 \approx 1.4 \times 10^6$, and we take $B_2 = B_1^2$

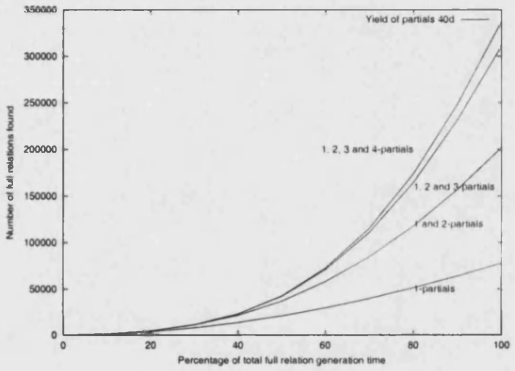

Figure 5-12: Yield of 1 – 4-partial relations – 40d

and $B_3 = B_1^3$. Using the Waterloo variant means we are testing values of around 20 digits for smoothness, and as $B_3 \approx 10^{18}$ here, we find that every value tested is 'at least' a 2-partial (i.e. if not it is either a 1-partial or a full). Thus we recover a lot of partial relations, but do not see that much benefit from combining the 3 and 4-partials. Taking a smaller smoothness bound $B_1 \approx 6 \times 10^4$, however, gives a very different picture (figure 5-13). The left hand graphic shows the overall yield of the partial relations – notice now that the yield from the 3 and 4-partials dwarfs that from the 1 and 2-partials. On the right, we take a closer look at the y-axis, and see an explosive growth in yield similar to that described by Dodson and Lenstra [38].

In figure 5-14 we look at the progress of cycle finding. For both of these log plots, we see essentially four phases. The first is an initial stage of quadratic growth, representing yield of the 1-partials, with little contribution from higher order partials. Yield then becomes cubic as the 2-partials begin to take effect – again, higher order partials do not contribute much to the total yield in this second phase. We then see the effects of the explosion – for 1,2 and 3-partials for this dataset, yield is growing as x^8 for x

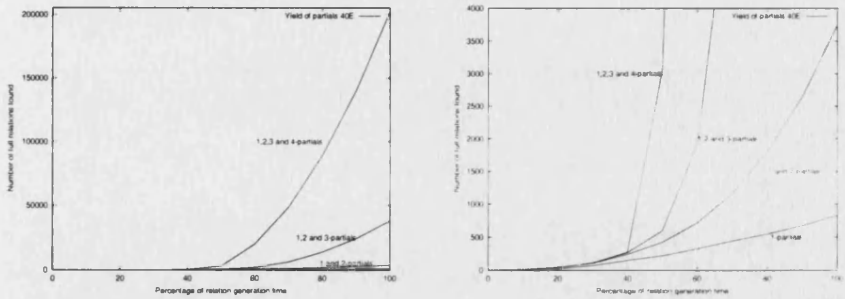

Figure 5-13: Yield of 1 – 4-partial relations – 40E

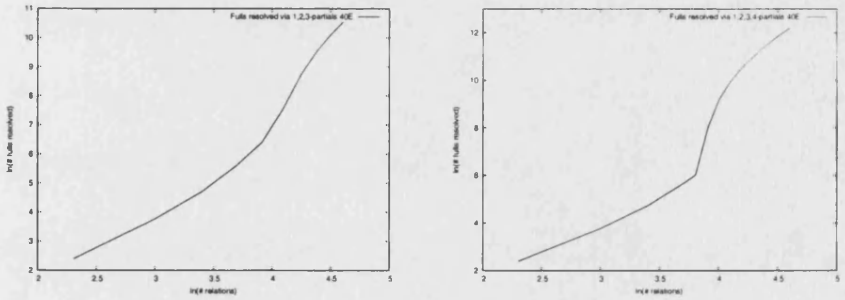

Figure 5-14: Yield of 1,2 and 3-partials (left) and 1,2,3 and 4-partials (right) – 40E

partial relations, while for 1,2,3 and 4-partials it is more like x^{17}. Finally, the explosion tails off and yield is roughly quartic for the final part of each plot.

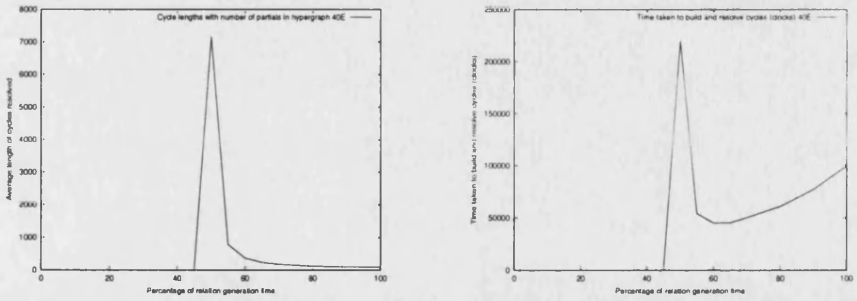

Figure 5-15: Length of hypercycles (left) and time taken to resolve (right) – 40E

As in [38], we also see that the 'explosion' is accompanied by a severe increase in the average 'length' of hypercycles. Figure 5-15 shows (left) the average length of hypercycles found, using the same dataset as in figure 5-13. This increase in hypercycle length in turn causes the actual time taken to build and resolve these cycles to increase (right). However, if we continue adding to the hypergraph, the average length of hypercycles reduces quite quickly. Time taken to resolve these hypercycles then also

drops, before the sheer number of cycles causes it to rise once more. The resulting full relations have considerably fewer entries, as noted by figures 5-16 and 5-17.

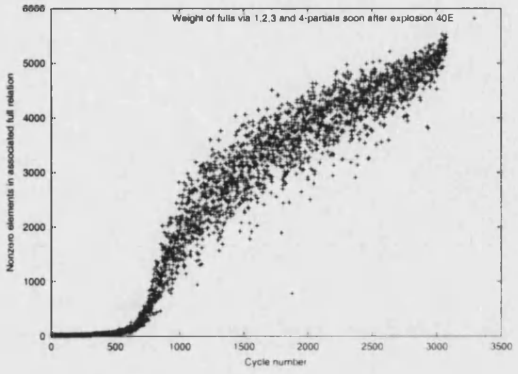

Figure 5-16: Density of fulls found immediately after explosion – 40E

In figure 5-16, we see that the increase in cycle length around the point of explosion is reflected in the density of full relations resolved. Had we continued building cycles, however, we would find that the average density of the ensuing fulls drops as more cycles are found, as shown in figure 5-17. Here we show the number of nonzeros per

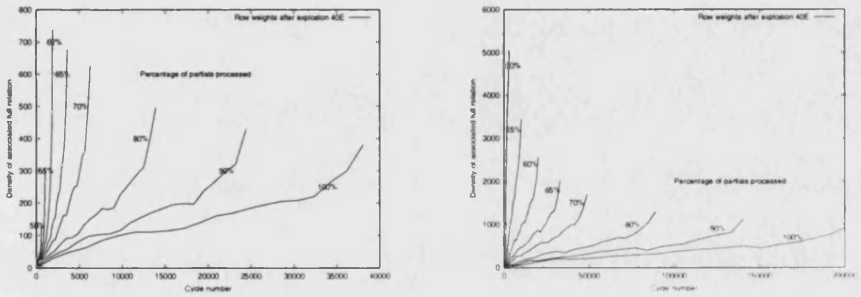

Figure 5-17: Density of fulls resolved from hypercycles – 40E

relation for full relations resolved from 1, 2 and 3-partials (left) and 1, 2, 3 and 4-partials (right). Thus, had we processed 50% of the 1, 2, 3 and 4-partial relations for this dataset, the 3000 or so resulting fulls would have had an average of some 2600 nonzeros. Had we processed 60%, this average would have dropped to around 1000 nonzeros, while processing all relations would have lead to fulls with an average of around 400 nonzeros per row. Processing 1, 2 and 3-partials only, we see a similar effect, although the 'explosion' comes with slightly more relations than before. It therefore pays to go *beyond* the 'big bang' (the point at which yield rapidly increases) in resolving partial relations with more than two large primes, as the resulting full relations are considerably less dense and consequently take less time to resolve[4].

[4]Compared with other full relations, of course, these are still far more dense – for this dataset,

5.4.4 Cost of resolving n-partials

As in the previous chapter, we must consider the effort required to process partial relations, starting with relation generation and the additional costs of storing partial relations, and subsequently the time taken to resolve these into further fulls.

Data generation

The cost of using partials is now slightly different – we can obtain all 1-partials *and* a certain amount of 2-partials (referred to in table 5.5 as '(1,1)'-partials) very cheaply, by checking for 1-partials in each of the two smoothness tests. Timings for generation of such data is given in table 5.5. In order to again highlight the practical speedup of the Waterloo variant, we include the time taken to generate identical datasets to those used in the previous chapter (timings for the generation of which were detailed in table 4.8). For these datasets, we see a 12-fold speedup when generating fulls for dataset 20d, a 23-fold speedup for dataset 25c, and a 44-fold speedup for dataset 30b. Again we see the cheapness of collecting 1-partials, and now we also obtain a certain amount of our 2-partials cheaply, as (1,1)-partials.

Dataset	Fulls	Time	+ 1-partials	+ (1,1)-partials	Time
20d	500	409	2,588	3,456	417
25c	500	3,235	4,903	11,002	3,259
30b	500	20,328	4,231	8,843	21,063
35A	500	531,518	16,956	139,229	534,284
40d	500	1,146,133	10,552	54,973	1,150,413

Table 5.5: Timings for generation of 500 fulls with and without storing of 1-partials on each smoothness test

The other 2-partials introduce the cost of a single primality test and Pollard ρ computation on one of the smoothness tests, as do the 3-partials. Finally, 4-partials are the most costly due to the need for two primality tests and Pollard ρ computations. Timings are shown in table 5.6, following on from table 5.5. Here, the 2-partial column shows the total number of 2-partial relations – both the (1,1) and the 'true' 2-partials. As before, we see the cost associated with identifying and storing true 2-partials.
Before resolving these relations, we again prune the datasets to speed subsequent processing. Pruning took the same approach as before – several passes without collision resolution, followed by a final processing with collision resolution. Timings – processing 1,2,3 and 4-partials together – are shown in table 5.7. Since pruning is proportional

the average full has some 15 nonzero entries, fulls via 1-partials have around 24, and fulls via 1 and 2-partials (cycles of length 3 or more) have around 45, compared with 160 and 405 nonzeros for fulls via 1, 2 and 3-partials and 1, 2, 3 and 4-partials respectively.

Dataset	+2-partials	+ 3-partials	+4-partials	Time
20d	3,638	28	0	492
25c	11,489	1,754	76	8,081
30b	11,066	5,204	661	73,666
35A	164,532	281,855	133,626	2,040,296
40d	59,879	55,655	13,814	3,779,748

Table 5.6: Timings for generation of 500 fulls with and without storing of 2-partials on each smoothness test

to the number of relations processed and the number of large primes they contain. it obviously takes more time as we move on to 3 and 4-partial relations.

Dataset	n-partials in	n-partials out	Time to prune	Passes taken
35A	6,755,792	2,150,738	101,690	9
40d	34,434,127	967,174	243,091	14
40E	9,511,872	671,286	88,610	15

Table 5.7: Timings for pruning of n-partial datasets

Here we show two separate 40 digit datasets, having slightly different initial parameters. Each have $g \equiv \sqrt{2} \bmod p$, since 2 is not a generator for $(\mathbb{Z}/p\mathbb{Z})^*$ in this example, but $p-1 = 2q$ for q prime. Dataset 40d has a very large smoothness bound - 1.4×10^6 - again with a view to generating a large linear system. However, this consequently generates a huge amount of partial relations. Dataset 40E has the much smaller smoothness bound 60,000.

1 and 2-partials

Resolving 1-partials is exactly as it was previously, but for completeness we show timings for the datasets under consideration in table 5.8. Again, we see that for very

Dataset	1-partials	Pruned	Fulls	ρ	Time to resolve
35A	166,925	18,507	10,604	21.06	87
40d	2,562,564	142,574	77,713	20.20	695
40E	58,962	1608	832	23.60	9

Table 5.8: Timings for resolving 1-partials

little effort we can gain a substantial number of full relations. For comparison of the density of the resulting full relations, we note that fulls found directly had on average 13.29 nonzero entries for dataset 35A, 12.69 for dataset 40d and 14.41 for dataset 40E.

Concerning 2-partials, we now have two different ways of resolving - the graph approach of Lenstra and Manasse [75] or the 'reduction' approach of Leyland et al. [81] and Dodson and Lenstra [38]. Timings for both techniques are shown in table 5.9. We no longer distinguish between the types of relation input to the procedure, since all 1 and 2-partials are processed together.

Dataset	Relations (pruned)	Fulls	ρ via Lenstra	Time via Lenstra	ρ via Leyland	Time via Leyland
35A	203,621	100,514	34.86	2,089	35.79	2,915
40d	479,050	201,992	27.84	3,942	27.95	4,070
40E	12,096	3,742	40.47	116	40.79	133

Table 5.9: Timings for resolving 1 and 2-partials

It is interesting to note that, for our implementations at least, the method of Lenstra and Manasse [75] was rather faster than the method of Leyland et al. [81]. However, as these are two different programs, much of this could be simply due to implementational inefficiencies in the latter case. As mentioned, the 'reduction' approach does give slightly longer cycles than the graph-theoretic method, although in practice we found this difference to be very small – in the above example for dataset 40E, cycles built by the former method contained an average of 4.25 edges – resulting in 40.79 nonzeros per full – while those built via the latter approach had 4.18, resulting in 40.47 nonzeros per full relation.

3 and 4-partials

We now use the methods described in this chapter to resolve 1,2 and 3-partials together, as shown in table 5.10, and subsequently 1,2,3 and 4-partials together as shown in table 5.11. Times refer to the time to both build and resolve hypercycles. In general, some 70% of the time taken to build the hypercycles was spent in setting up hash tables to index relations by the primes they contain, and vice versa.

Dataset	Relations (pruned)	Fulls	ρ	Time to resolve
35A	1,131,825	570,731	63.19	32,426
40d	858,718	311,182	38.05	11,702
40E	149,174	38,067	160.47	5,841

Table 5.10: Timings for resolving 1,2 and 3-partials

When taking into account the time required to prune the datasets, we see that processing 3-partials takes considerably more effort than does processing 1 and 2-partials only. We are now rather more hampered by the small dataset sizes we are considering here

– the vast amount of data collected makes it rather harder to see exactly how practical the technique is. It is a similar story when processing the 4-partials.

Dataset	Relations (pruned)	Fulls	ρ	Time to resolve
35A	2,150,738	1,146,379	82.21	88,786
40d	967,174	337,721	41.80	14,227
40E	671,286	202,273	405.86	103,297

Table 5.11: Timings for resolving 1,2,3 and 4-partials

We note that in almost all our experiments, we obtained exactly k fulls from k hypercycles, where $k = E - V$ and E and V correspond to the number of relations and distinct large primes in the hypergraph respectively. The only exceptions to this were firstly when processing a very small proportion – some 5% or so – of the partial relations, and secondly, as before, in the case of isolated loops.

Overall speedup

We have seen that one can obtain a large amount of full relations via the use of 3 and 4-partials. It is not yet clear, however, whether the time taken to store and process these partials allows an overall saving in processing time. We now consider this question. As before, we try to estimate the timings we can save by gathering and resolving partials, compared to the time to generate full relations directly. Here we consider dataset 40E, but in order to give a more realistic example, we cut the double large prime bound from B_1^3 to $100B_2 = 100{B_1}^2$ to create dataset 40Es. B_1 here remained at 60,000. Looking at the yield of the various partial relations gives us figure 5-18.

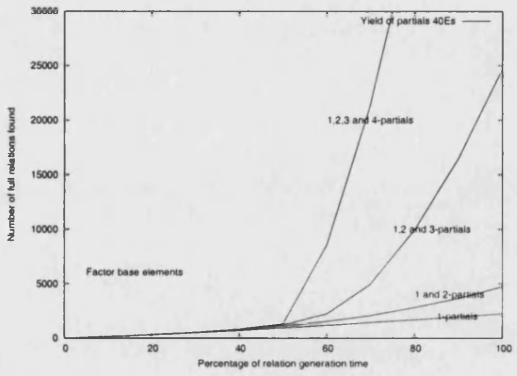

Figure 5-18: Yield of 1 – 4-partial relations – 40Es

We see firstly that, as in the last chapter, cutting the large prime bound does not result in a commensurate decrease in yield (compare with figure 5-13). We now look at the

points at which we could have terminated relation generation, were we using partials. Using 1,2,3 and 4-partials, we see that we could have stopped relation generation after some 60% of the total time we actually spent. We would then have a total of 8,537 fulls in 6,058 unknowns.

Stop point	Fulls (direct)	Fulls (via partials)	Time to generate	Time to resolve	Ave. ρ for 8,537 fulls
60%	864	7,673	10,505,436	20,388	757.0
70%	1,008	20,488	12,256,343	20,737	172.9
100%	1,441	80,220	17,521,220	32,045	54.0
•	8,537	•	101,313,103	•	14.4

Table 5.12: Savings using 1,2,3 and 4-partials - 40Es

Table 5.12 shows the time taken to build and resolve full relations, using 1,2,3 and 4-partials. From figure 5-18 it would appear that terminating relation generation after around 60% of the total time we actually took would have been more efficient. The timings back this up; but we also show the advantage of continuing relation generation rather further. The advantage this gives is an order of magnitude drop in the density of the average full relation. Even allowing for this extra processing, we see that we are still well within the time it would have taken to generate the equivalent number of full relations directly.

Stop point	Fulls (direct)	Fulls (via partials)	Time to generate	Time to resolve	Ave. ρ for 7,222 fulls
75%	1,080	6,142	12,902,190	3,417	139.8
80%	1,152	8,780	13,762,336	4,131	133.7
100%	1,441	23,292	17,214,866	6,650	110.9
•	7,222	•	85,707,301	•	14.4

Table 5.13: Savings using 1,2 and 3-partials - 40Es

Looking at only the 3-partials in figure 5-18, we find that the point at which we obtain a sufficient number of relations occurs after some 75% of the time taken. As shown by table 5.13, again it is in our interests to go rather further than this point in order to produce a lighter set of full relations (although here the difference is not so pronounced). By not allowing more than one application of Pollard ρ (and associated primality test) per attempt, we also take slightly less time in relation generation.

Simply using 1 and 2-partials for this dataset would not have given us enough fulls in the time we took to generate these relations, so the use of 3 or 4 large primes does give a practical speedup in this case. We emphasise, however, that we are using a slow method of relation generation – for a faster method using sieves, early abort and other strategies, it is unlikely that we would benefit from using two or more large

primes at the kind of scales under consideration here. The implementation of Cavallar [21] compares a '3+2' and a '2+2' large prime NFS implementation to factor numbers of 179 digits, and concludes that no advantage was to be gained by using more large primes, but for the fact that a rather smaller factor base could have been used. Leyland et al. [81], on the other hand, estimates a speedup by a factor of 1.7 when using the Multiple Polynomial Quadratic Sieve with three large primes, compared to using the same method with two large primes to factor a number of 129 digits.

5.5 Summary

In this chapter we have investigated how the techniques of Dodson and Lenstra [38] and Leyland et al. [81] for resolving 3-partials for factoring purposes carry across to the discrete logarithm case. We generated such data by means of the Waterloo variant of the basic index calculus method. This forces us to take account of exponents of large primes, and we showed how the techniques of the previous chapter to resolve 1 and 2-partial relations can be simply adapted to deal with this situation.

We illustrated the potential difficulties of resolving relations involving more than two large primes, and have shown how, in general, one can resolve all hypercycles built, such that the yield is again all but ε of that in factoring applications. The exceptions are again isolated odd cycles, which are usually some kind of loop. The downside of this technique is that it can create slightly longer cycles, leading to more nonzero elements in the ensuing fulls; so if one is using a maximum of two large primes per relation, it may be advisable to use the graph theoretic method of Lenstra and Manasse [75], or else some strategy to minimise the weight of resulting fulls, such as that of Cavallar [20]. We found empirically that the approximation $E - V$ actually gave us the exact number of cycles resolved by the procedure of [81] so long as we processed a reasonable amount of the partial relations; and this seems a good practical indicator of potential yield during relation generation.

We found experimentally that the yield of 1-partials grew quadratically with the number of relations processed, while the number of fulls obtained from processing 1 and 2-partials grew with the cube of the number of partials processed. Processing higher order partials resulted in explosive increases in yield in a similar manner to that of the factoring method described in Dodson and Lenstra [38]. Practical speedup is not easy to judge for the small examples presented here, but it is reasonable to assume that for discrete logarithm computation modulo a k-bit prime, the technique will be as effective as it is when applied to factoring a k-bit number.

Part III

Linear Algebra and Computation

Chapter 6

Linear Algebra

In this chapter we consider the final stage of precomputation for the index calculus method, and look at methods for solving systems of linear equations over a finite field. We discuss the effect of using a preconditioning method – structured Gaussian elimination – coupled with the iterative Lanczos algorithm, and examine how these methods are affected by the techniques of the previous chapters.

6.1 Solving linear systems

Index calculus methods, both for discrete logarithm computation and for factoring, require us to solve a large system of linear equations

$$Ax = b$$

where A is an $m \times n$ matrix, b is a known m-vector and x is an unknown n-vector of the discrete logarithms of our factor base elements. By 'large' here we are talking of up to 10^6 or more equations in some 10^6 unknowns. For factoring purposes, we are solving a homogeneous system modulo 2, while for discrete logarithm computation modulo a prime p we look to recover a full solution vector modulo the prime factors of $p-1$; although in practice we may not recover a solution value for every unknown in the system. Since these systems are to be solved over a finite field, a certain amount of complexity is added to standard algorithms, and some are even rendered useless. Conversely, the systems under consideration do have a certain structure that we may use to our advantage. Several studies involving use of index calculus-type methods discuss the linear algebra step, as it is very much a practical bottleneck in the procedure due to the difficulty of parallelising available techniques. LaMacchia and Odlyzko [70] and Pomerance and Smith [109] identify several effective techniques for the solution of sparse linear systems modulo a prime. These include iterative schemes such as the Lanczos, Conjugate Gradient and Wiedemann algorithms, with or without a preprocessing step

involving some kind of Gaussian elimination. LaMacchia and Odlyzko [70] advocate the use of modified or structured Gaussian elimination (SGE) followed by use of the Lanczos or Conjugate Gradient algorithm. Preconditioning strategies for the linear algebra step (from a factoring viewpoint) are extended and improved by Cavallar [20].

6.1.1 What is the matrix?

In order to make the linear algebra step more efficient, we try to take advantage of the structure of the system. We can note certain properties about the system.

Firstly, and most importantly, it is extremely sparse. As an example, for one of our datasets of some 25,000 unknowns, the average row contains around 9 nonzero elements. Even with around 10^5 unknowns, there are generally no more than 50 nonzero coefficients in each row of the matrix [70]. We can thus save on memory by using a sparse encoding.

Secondly, although random numbers were used during the generation of the rows of the matrix, the resulting system itself has a certain amount of structure – the matrix is considerably more dense toward the left hand side, i.e. more nonzero coefficients exist for the smaller factors in our factor base[1], as noted in chapter 3. Nonzero values corresponding to the larger primes are generally ± 1. In all datasets studied as part of this investigation, over 60% of the nonzero values in the matrix occur in the first 5% of the columns. This causes immediate heavy fill if 'traditional' Gaussian elimination is used, and any benefits of the sparse structure are quickly lost.

Finally, the system is usually underdetermined. We generally require more rows than columns in our matrix in order to maximise the number of unknowns for which we can recover a solution (due to the random nature of relation generation). For the smaller systems discussed in this chapter, in general some 5-6% of the primes in our factor base do not occur in any relation. Taking more rows will usually help reduce this loss, but will of course increase the size of the system. Conversely, since not all factor base elements will be represented, we can generally get away with generating slightly fewer relations – Lenstra and Manasse [74] note that, for a factor base of n elements, we generally need only generate some $0.99n$ relations.

Most discussions concerning discrete logarithm computation in the literature end after the linear algebra step – actual computation is rarely discussed. However, this last point is very important. For factoring purposes, we take excess rows purely to guarantee several linear dependencies, since finding only one may simply return the trivial factorisation. We wish, though, to minimise both the size and density of the matrix to speed processing. The index calculus method for discrete logarithms and its analogue

[1] If using a 'multiple factor base' approach such as NFS or FFS, of course, such a structure would arise for each factor base. It is straightforward however to permute columns by weight to restore a simple structure.

for factoring now diverge, however. While it remains the case that we wish to minimise the time taken to solve the linear system, for the discrete log case we also want to solve for as many of the factor base elements as possible, to maximise the size of our database of known values. In general, not all factor base elements are represented in the matrix. Using a certain amount of denser relations will allow us to increase this coverage of values, but will generally compromise solution time.

Even when taking many more rows than we have factor base elements, the system will generally remain underdetermined. As shown by figure 6-1, plotting the percentage of unknowns which remain unresolved after solving against the amount of excess rows contained in the system (expressed as a percentage of the number of columns in the system), we see that even taking an excess of 50% does not allow us to compute all the unknowns in the system. Some 2% remain.

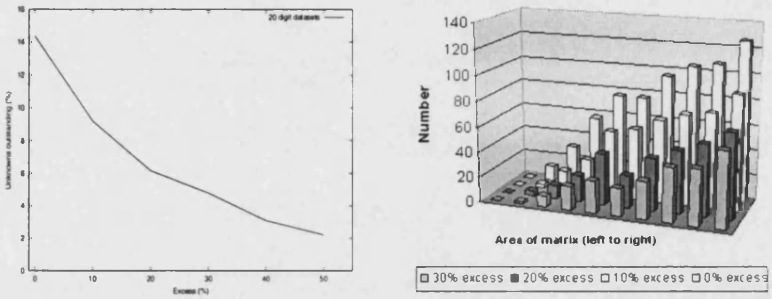

Figure 6-1: Unknowns outstanding (left) and their distribution - 20d

This difficulty in obtaining a full solution vector has important consequences when using these results in the final stage of the index calculus procedure, and we wish to minimise this loss if possible. We note, as illustrated by the right hand figure, that the majority of these outstanding values are found toward the right hand side of the matrix – the sparse area of the system, corresponding to the larger primes in our factor base, which occur in few, if any, relations.

6.1.2 Gaussian elimination

Traditional Gaussian elimination is often the first port of call for anyone wishing to resolve a system of linear equations. The algorithm in its most basic form is fairly simple to implement, and is most commonly used to achieve so-called LU decomposition. For our purposes, however, we do not need to go quite so far – we could simply put the system $Ax = b$ into upper triangular form, and then use back substitution to recover the unknown values. We can accomplish this by means of elementary row operations: we may swap rows, multiply rows by some constant, and add or subtract rows.

The fundamental problem with standard Gaussian elimination, particularly for the

large systems we are considering, is the potential level of *fill*. Subtracting one row from another will create new nonzero entries in the matrix, and an initially sparse matrix may quickly become very dense as the elementary row operations of Gaussian elimination are computed. This obviously inhibits the practical performance of the algorithm both in terms of time taken for subsequent operations and of course in the amount of space needed to store the matrix. Since to eliminate the first column takes $n(n-1)$ operations, the complexity of Gaussian elimination is $O(n^3)$. However, by taking account of the structure of the system, in practice, an index calculus-type system can be solved in time $O(n^{2+\varepsilon})$, as noted by Odlyzko [99].

6.1.3 Structured Gaussian elimination

A number of people have proposed *intelligent* or *structured* Gaussian elimination (SGE) (see Bender and Canfield [11], LaMacchia and Odlyzko [70], Pomerance and Smith [109]) to try to take advantage of the sparsity and structure of the system. The idea is to output a smaller system to solve with another method – possibly standard Gaussian elimination – and then we can use back substitution to recover the other values.

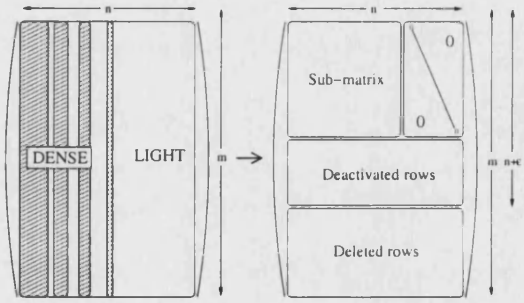

Figure 6-2: Aims of structured Gaussian elimination

The algorithm has no exact definition, but basically involves the same elementary row operations used in traditional Gaussian elimination. One tries to apply these operations in a manner that will minimise fill. The routine outputs two smaller systems of equations as shown in figure 6-2. We input an $m \times n$ matrix. We then carry out elementary row and column operations to permute the matrix such that the top $k + \delta$ rows contain nonzero values in only the first k columns. These rows then form a $(k + \delta) \times k$ submatrix. The subsequent l 'deactivated' rows – up to row $n + \varepsilon$ – have been removed from the system during processing, and can be ordered such that the span of columns for which rows contain a nonzero value increases by precisely 1 as one considers successive rows. Finally, any further rows are discarded (or can be retained for further back substitution if necessary).

The smaller submatrix may be solved by another method – in our case the Lanczos algorithm – and then the rows marked 'deactivated' may be solved by back substitution

using these results. Note that this diagram is *not* to scale – our input matrix is very much more biased toward the left hand side, and in reality we can expect our submatrix to be around 5% of the size of the original matrix, or indeed even smaller.

The following version[2] of the SGE algorithm is based on that described by Bender and Canfield [11] – itself following Pomerance and Smith [109] – with the addition of step 1c to remove excess rows. Bender and Canfield describe the method as applied to data over GF(2) arising from factorisation calculations.

The algorithm splits the matrix into *active* and *deactivated* regions and uses the concept of *heavy* and *light* columns and rows, where 'heavy' indicates a large number of nonzero entries in the given column or row. Similarly we define the *weight* of a row or column to be the number of nonzeros it contains. For the purposes of the SGE algorithm (algorithm 8), we consider the weight of a column or row as the number of nonzero values it contains in the *active* part of the matrix only.

Algorithm 8 Structured Gaussian elimination

Input: $m \times n$ matrix, m vector
Output: $k + \delta \times k$ submatrix, l rows for back substitution
Step 0 – Declare fraction of columns to be 'inactive'
Step 1 – Initial clean up:
repeat
1a Discard columns of weight 0
1b Store columns of weight 1 and their corresponding rows
1c Delete an excess row
until all columns have weight 1
Step 2 – Deactivation:
repeat
2a If a row has weight 1, eliminate it and store this row and the column intersecting it
2b Deactivate a column and repeat step 2a
until all columns are deactivated or eliminated

The first step (here referred to as step 0) assigns an initial partition to the matrix, and we look to preserve sparsity in the 'active' region over the course of the two subsequent phases of the algorithm. It should be noted that this partition of columns is not fixed for the duration of the algorithm; rather, it will generally shift to the right as the algorithm progresses. Note further that the only step which will cause any fill is step 2a, the Gaussian elimination step, and that this fill is restricted to the 'deactivated' area of the matrix. Steps 1a and 1b are iterated before step 1c is invoked, and we subsequently iterate steps 1a and 1b again. We maintain a certain excess δ in the submatrix, again to guard against any linear dependencies which may be present. We may subsequently solve the submatrix consisting of the deactivated columns and rebuild the solutions of

[2]Note that various versions of the algorithm are possible. Once actions which minimise the amount of fill in the matrix are identified, the order of their application may vary.

the full matrix by means of the rows eliminated in steps 1b and 2a. The number of columns initially declared 'inactive' thus goes some way toward defining the size of the output submatrix.

6.1.4 Lanczos algorithm

Once the system has been processed with SGE, we need to solve the resulting small system. The Lanczos algorithm [72] for solving systems of linear equations first appeared in 1952, and is particularly useful for solving sparse systems. It was not originally intended to be used over finite fields, but it has been successfully adapted to deal with the additional complications these situations present. Here we use the algorithm as a 'black box' to solve the system output from SGE, but for completeness we give a brief description. The following is from LaMacchia and Odlyzko [70][3].

The Lanczos algorithm attempts to provide a vector x as a solution to the system $Ax = w$ where A is a symmetric $n \times n$ matrix and w is a known column vector of dimension n. An initial problem, then, is that in general we will have $m \times n$ matrix B $(m > n)$. In order to get round the requirement for a symmetric input matrix, let D be an m x m diagonal matrix with elements chosen at random from $(\mathbb{Z}/p\mathbb{Z})^*$. Then let

$$A = B^T D^2 B$$

and

$$w = B^T D^2 u$$

To solve the system $Ax = w$ the Lanczos algorithm proceeds as shown in algorithm 9. This recursive procedure continues until for some $j \leq n$ the algorithm finds a w_j for which $(w_j, Aw_j) = 0$; i.e. w_j is conjugate to itself. If $w_j \neq 0$ then the algorithm has failed. However, if $w_j = 0$ then

$$x = \sum_{i=0}^{j-1} b_i w_i$$

is a solution to $Ax = w$, where

$$b_i = \frac{(w_i, w)}{(w_i, v_{i+1})}$$

It may be the case, depending on our method of relation generation, that we wish to solve a homogeneous system $Ax = 0$. In this case, we can compute $w = A.r$ for some random vector r, and solve the system $A.x' = w$. We can then compute $x = x' - r$ to recover the desired solution to $A.x = 0$.

[3]LaMacchia and Odlyzko [70] apply the *Conjugate Gradient algorithm* to solve the linear system modulo 2, whilst the Lanczos algorithm is used to solve the system mod $(p-1)/2$ (the Lanczos algorithm, when applied modulo 2, will terminate with a self-conjugate vector 50% of the time on average and is thus not practical). The conjugate gradient algorithm was not implemented as part of this study.

Algorithm 9 Lanczos algorithm

Input: $n \times n$ matrix A, n vector w
Output: n element solution vector x

Initialise:
$w_0 = w$
$v_1 = Aw_0$
$w_1 = v_1 - \frac{(v_1,v_1)}{(w_0,v_1)} w_0$
$b_0 = \frac{(w_0,w)}{(w_0,v_1)}$
$x = b_0 w_0$
Process:
for $i > 1$ **do**
  $v_{i+1} = Aw_i$
  $w_{i+1} = v_{i+1} - \frac{(v_{i+1},v_{i+1})}{(w_i,v_{i+1})} w_i - \frac{(v_{i+1},v_i)}{(w_{i-1},v_i)} w_{i-1}$
  if $(w_j, Aw_j) = 0$ **then**
    if $w_j = 0$ **then**
      return x
    else
      Failure
    end if
  else
    $b_i = \frac{(w_i,w)}{(w_i,v_{i+1})}$
    $x = x + b_i w_i$
  end if
end for

When applied to a finite field, the Lanczos algorithm has none of the rounding errors that would potentially cause instabilities. We do though have the problem that, in a finite field, one can find a vector which is self-conjugate but is itself nonzero. In practice, however, no execution of our implementation failed for this reason.

Again we may look at the structure of the matrix in order to increase efficiency of the implementation. The algorithm calls for the computation of several inner products. We have the advantage that it is not actually necessary to compute and store the (dense) matrix A – we can compute for example Aw_i as $B^T(D^2(Bw_i))$. Furthermore, we know that the majority of the nonzero values in the matrix are small; often ± 1. However, taking advantage of the sparse nature of the matrix by storing it in some kind of compressed format as in the structured Gaussian elimination routine does lead to increased overhead in accessing memory when computing inner products and matrix multiplications. The runtime of the Lanczos algorithm can be shown to depend on ne (see LaMacchia and Odlyzko [70]), where n is the number of columns in the input matrix and e is the number of nonzero entries in this matrix.

6.1.5 Others

The Lanczos algorithm is one example of a Krylov-subspace method. Other such methods exist, such as the Conjugate Gradient (Hestenes and Stiefel [54]) and Wiedemann (Wiedemann [137]) algorithms. These methods may of course be applied directly to the full system without preprocessing via SGE. For an in-depth discussion of these techniques, see Lambert [71]. Various methods for parallelising these methods have been proposed. Montgomery [90] shows how, when solving modulo 2, one can process N vectors simultaneously, where N is the computer word size. Contini [25] sketches a master-slave model for the Lanczos algorithm, where each slave holds some subset of the rows of the matrix. The Wiedemann algorithm has been block parallelised by Coppersmith [27]. At present, it seems that block Lanczos and block Wiedemann are the method of choice for solving large linear systems originating from index calculus based methods; usually with a preprocessing step of SGE to reduce the size of the system whilst maintaining, as well as possible, some degree of sparsity.

6.2 Experimental results

Both a structured Gaussian elimination routine and the Lanczos algorithm were implemented as part of this study. Our intention was to investigate the impact of both generator choice and the use of 'large prime' derived relations on the linear algebra step, but also to discuss the workings of the SGE procedure in a little more depth, in the hope of clarifying the procedure for others wishing to implement the technique.

6.2.1 SGE

The execution of the SGE procedure is best illustrated diagrammatically as below. Here we are solving modulo $q = (p-1)/2$.

SGE phase 1

Figure 6-3 illustrates the application of phase 1, the 'initial clean up', to a 4801 x 4801 matrix resulting from 20 digit discrete logarithm calculation. The y-axis plots the value of ne where n is the number of columns in the submatrix output by SGE (an estimate of complexity for the Lanczos algorithm to solve this submatrix system), and e is the number of nonzero entries in this submatrix.
There is an immediate fall in complexity ne as the operations of steps 1a and 1b eliminate many columns thanks to the initial deactivation. Complexity then decreases smoothly to the end of the first phase. It is worth noting that it can be useful to keep the rows we 'remove' in step 1c in a file - we may find that they allow us to recover more unknowns in the back substitution phase of the full solution process. If we take a

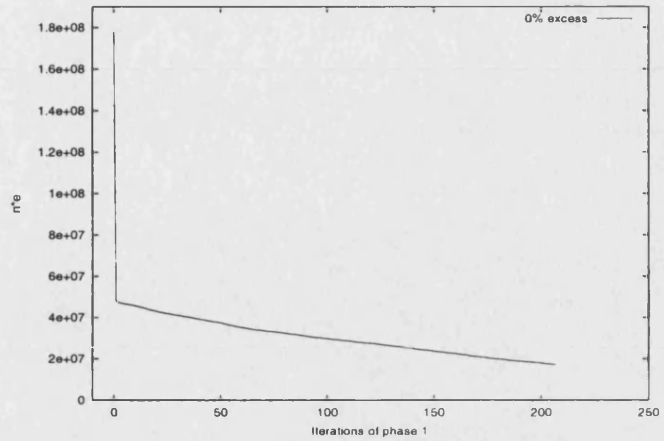

Figure 6-3: SGE Phase 1 (0% excess) – 20d

larger number of excess rows, we create more work for the first phase of the algorithm, due to having more rows to remove. However, this can actually give us a smaller system to work with going into phase 2.

SGE phase 2

Carrying out a similar examination of the second phase of the algorithm, we obtain figure 6-4.

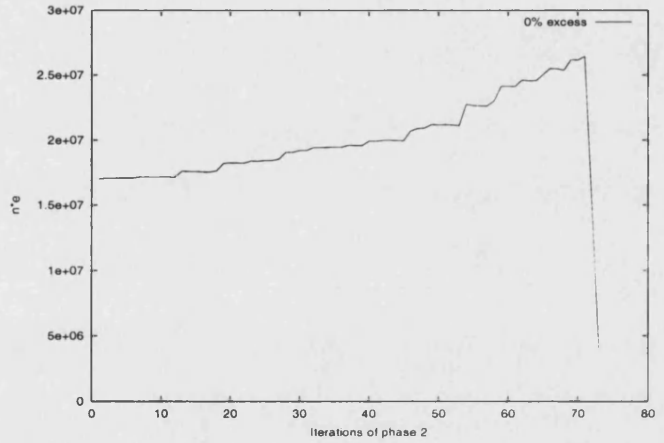

Figure 6-4: SGE Phase 2 (0% excess) – 20d

With the operations of step 2a we begin to see the effects of fill which causes the complexity to rise in spite of the reduction of the absolute size of the submatrix. There comes a point, however, when a certain number of columns have been deactivated or eliminated, at which the system collapses – the 'created catastrophe' of Pomerance and

Smith [109] – and the algorithm terminates. What motivates this collapse? Figure 6-5 plots the number of rows eliminated by successive applications of step 2a for the first dataset.

It can be seen in figure 6-5 that we have a near linear increase followed by a dramatic removal of all remaining columns without further application of step 2b. We have been deactivating columns based on their weight, so at this point we are well into the sparse side of the matrix. In the datasets tested, over 70% of the nonzero entries occurred in the first 10% of the columns. There comes a point when elimination of a single row and column allows removal of all others due to their sparsity. For the dataset illustrated in figure 6-5, collapse occurs when some 80% of the columns in the matrix have been either eliminated, deactivated or removed as zero columns. When one considers the 20 digit dataset having 20% excess rows, one finds that collapse occurs when 90% of columns have been removed in one of these ways. Comparing the collapse point across datasets we see that it occurs with fewer iterations of step 2a when we have a greater excess of rows to columns - we have shifted more work onto the first phase of the algorithm.

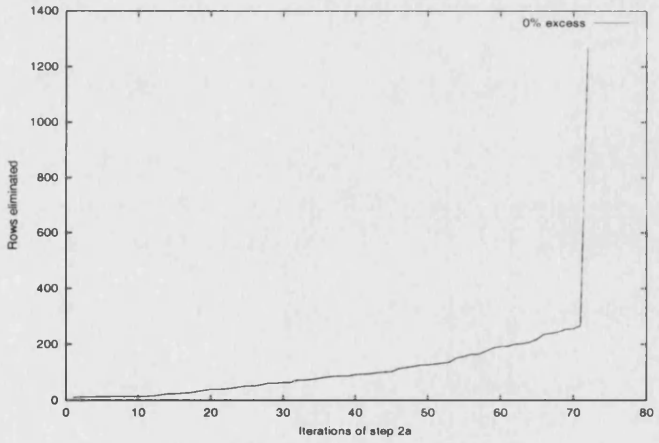

Figure 6-5: SGE Phase 2 collapse (0% excess) – 20d

6.2.2 Observations

We now make several observations concerning the operation of the SGE procedure. Ideally we would like to give estimates for optimal parameters with which to run the SGE procedure, however, as noted by LaMacchia and Odlyzko [70], this varies from one dataset to another. We can however note any trends which it would be useful to be aware of when using this method.

Blowup

It was observed that a 'blowup' of coefficients could occur during step 2a (the elimination step), causing overflow errors when C++ 'long' data types were used for nonzero matrix entries. This is also noted by Gordon and McCurley [49]. Although most nonzero entries in the matrix are ± 1, one can have other values. Indeed, it is not uncommon amongst the columns representing lower factors to find exponents as large as 10 or 12. Using large prime variant techniques can cause even larger values to appear. When applying step 2a, these large values can propagate and, in addition to encountering the effects of fill, we also get much larger coefficients.
It is desirable, for both memory and time reasons, to use 'long' data types (at most) for the nonzero matrix values, but if the initial parameters are not chosen with a degree of care, this blowup effect can cause coefficients to overflow a long data type and multiple precision data types become necessary, severely reducing performance. For the same reasons, we do not wish to use multiple precision values for the matrix exponents and reduce modulo q, instead of using C++ long data types and using 'fraction-free' row elimination. Values in the matrix can be negative, particularly if one is considering data from the Waterloo variant index calculus technique. Reducing these modulo q immediately introduces (for the datasets we are considering) 20 or 30 digit numbers into the matrix and performance is immediately and drastically reduced.
The smaller the initial deactivation, the more work the algorithm is forced to undertake in both phase 1 and phase 2. As a result of this, step 2a is applied to the extent that the nonzero values can overflow their data type maximum value. This could potentially be alleviated by maintaining a larger excess in the submatrix – which will reduce the work done by phase 2 – and then removing the heaviest rows (once the SGE algorithm has terminated) in order to bring the excess down to a more reasonable amount.

Choice of rows/columns

A second point for consideration is the question of which columns we should flag as 'deactivated', both in step 0 and in step 2b. The simplest method – as implemented here – is to choose the row/column with the highest weight. Other options are of course possible – if we deactivate a column which occurs in a row of weight 2, then by deactivating this column we can obtain a row with a single nonzero entry in the light columns, and immediately eliminate it from the system[4], as noted by Cavallar [20], Pomerance and Smith [109]. A similar idea can be applied when considering the removal of excess rows. We found it best to remove excess rows one by one rather than 'en masse' as is suggested by LaMacchia and Odlyzko [70], since otherwise we may

[4]For the dataset sizes examined here, we found that the increased book keeping needed to track this information negated any benefits it brought about.

effectively create a new excess due to having removed a number of unknowns from the system.

Excess rows

In addition to the number of excess rows we input to the procedure, it is also important to consider a certain excess as output from the SGE algorithm. If we simply remove all excess rows in step 1c we risk passing an under-determined submatrix to Lanczos and failing to find a full solution vector. The results of our tests indicate that an excess of up to 12% may be necessary, although in general a 5% excess sufficed to get a full result set from Lanczos. Of course, an incomplete result set from Lanczos can, in the worst case, cause all results obtained via back substitution to be incorrect.

Missing values

It was noticed that back substitution over the rows deactivated in steps 1b and 2a sometimes failed to resolve all outstanding unknowns – certain values, which were found if one applied Lanczos directly to the full system, could not be resolved when using the SGE→Lanczos→back substitution approach. In an attempt to remedy this, both rows deleted in step 1c and 'ancestor rows' from step 2a were also retained and additional back substitution over these rows yielded further values. However, it often remained the case that certain values could not be resolved. This comes from deleting the 'wrong' row in some sense over the course of the SGE algorithm. We may well have the situation where two particular rows are the only rows which can be used to resolve a certain number of variables. If we 'split them up' and both rows do not go to the Lanczos algorithm via the output submatrix, we may find that we cannot resolve two of these unknowns. Back substitution, which relies on the fact that we have a single unknown outstanding in each successive row, then also fails. Phase 3 depends upon our building a database of known values – any reduction in the number of these values will have a detrimental effect on discrete logarithm computation.

Optimal stop point

The overall goal of the SGE→Lanczos→back substitution model is to bring the matrix size down to a more manageable level, whilst attempting to preserve the sparse structure as much as we can. It is natural then to enquire when we should stop the SGE procedure. It is interesting to observe Lanczos complexity plotted against the number of columns deactivated in the initial 'partitioning' of the matrix, which is the major factor governing the size of the submatrix on termination of the SGE algorithm. We see in figure 6-6 that for certain datasets it is not in fact optimal to reduce the submatrix as much as possible, and, as noted by LaMacchia and Odlyzko [70], adding more rows

to the system may in fact allow faster overall solution time. The rather abrupt breakoff of these graphs is due to the effects of coefficient blowup mentioned earlier.

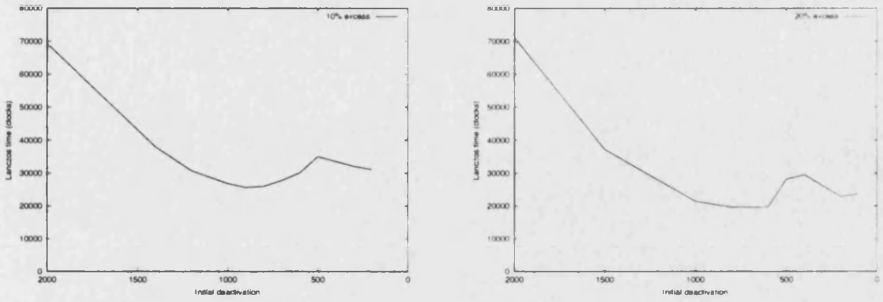

Figure 6-6: SGE initial deactivation v Lanczos time (10% and 20% excess) – 25c

This dataset comes from computation of discrete logarithms modulo a 25 digit prime, with 10% (left) and then 20% more rows than columns. It is clear that there comes a point when the increasing density of the submatrix begins to offset any further advantage gained from reducing its absolute size. Of course, the time taken by the SGE routine increases depending on both the input size of the matrix and the desired output size of the submatrix. We can thus save a certain amount of time by terminating the SGE algorithm prior to maximum reduction. Further, the effect of coefficient 'blowup' is reduced by earlier termination, thus keeping the absolute size of the matrix coefficients smaller.

Results indicate that a smaller absolute size of submatrix could be achieved through a smaller initial deactivation *if* blowup could be avoided. However, such a submatrix would be extremely dense (as much as 97% or more in our tests), which is undesirable. Regarding optimal initial deactivation, Pomerance and Smith [109] deactivate some 5% of the columns to form the initial partition of the matrix. In our implementation, we found that deactivating nearer 20% gave a minimum solution time for a square matrix, but this fell as one took more excess rows – optimal solution time came when deactivating around 10% of the columns of matrices having 30% or more excess rows.

6.2.3 Overall savings

We now turn our attention to the overall savings one can achieve with these methods. We compare the solution of a given system obtained via the Lanczos algorithm with equivalent results obtained via a structured Gaussian elimination/Lanczos/back substitution combination. Timings for the back substitution step are very small in comparison and so are omitted from table 6.1.

The interesting point to note is that, perhaps contrary to intuition, a larger system may actually be solved in less time if it has a sufficient amount of excess rows. This

SGE in	ρ (%)	Time	Lanczos in	ρ (%)	Time	Total
•	•	•	$22,710 \times 22,710$	0.04	3,555,873	3,555,873
$22,710 \times 22,710$	0.04	7,060	$4,187 \times 3,987$	0.84	258,066	265,126
•	•	•	$28,535 \times 23,377$	0.04	4,558,603	4,558,603
$28,535 \times 23,377$	0.04	7,998	$3,138 \times 2,988$	1.06	125,531	133,529
•	•	•	$34,290 \times 24,489$	0.04	5,489,076	5,489,076
$34,290 \times 24,489$	0.04	9,680	$2,607 \times 2,482$	1.17	76,588	86,268

Table 6.1: Time taken to solve linear systems – 30b

rather curious result is also noted by LaMacchia and Odlyzko [70]. In this case, we see that timings are still going down when we take 40% more rows than columns. The results of [70] suggest that this continues with larger datasets. In our tests, we saw a similar situation for the 25 digit data (where taking some 20% excess rows gave a minimum solution time), but not for the 20 digit data (here, a larger system simply took longer to solve).

Even with 25,000 unknowns there are still in general less than 10 nonzeros in the average row. The larger matrix is 'more sparse' than those we have considered for the 20 and 25 digit datasets if one represents sparsity as a percentage, and it would appear that the SGE algorithm can take a greater advantage of this when given a certain number of excess rows. If this pattern continues one could take a considerable number of excess rows for very large datasets, so long as the time required to generate these relations is offset by gains in time for the linear solve step. This excess gives a greater choice as to optimal rows to delete, but, of course, too great an excess could also offset this benefit by requiring more time to locate and manipulate the data within the SGE and Lanczos algorithms. As a final point, it is worth noting again that the savings here were obtained even with rudimentary implementations – they could without a doubt be made more efficient.

6.2.4 Generator choice revisited

We now consider the effect that the reduction in weight brought about by choosing our generator may have on our method of solving the linear system. As described in chapter 3, we attempt to choose our generator such that we may remove the column corresponding to factor base element 2 prior to the linear algebra step, with a view to cutting down on both storage and processing time[5].

[5]Certain factoring papers (notably that of Huizing [59]) advocate removing the k heaviest columns in the matrix prior to solution with a Block Lanczos approach. A full result set is subsequently reconstructed at a later point. Whilst this will obviously lighten the matrix still further, we can argue that removing the generator column does *not* require us to reconstruct the result for this factor base element, and thus would still be beneficial.

Generator choice and Lanczos

We first consider the slightly simpler case of the effect of generator choice on the Lanczos algorithm. As noted in chapter 3, for our 30 digit dataset, the columns corresponding to factor base elements -1 and 5 (which was the value of generator used for this dataset) account for some 8% of the total nonzeros in the system. Table 6.2 shows the time taken by the Lanczos algorithm to solve 3 systems – one square, one having 20% excess rows and one having 40% excess rows, both with and without the columns corresponding to factor base elements -1 and 5.

Matrix	Time to solve with -1 and g columns	Time to solve without -1 and g columns	Saving (%)
$22,710 \times 22,710$	3,555,873	3,362,390	5.44
$28,535 \times 23,777$	4,558,603	4,318,898	6.01
$34,290 \times 24,489$	5,489,076	5,198,347	5.30

Table 6.2: Time taken via Lanczos with/without g column - 30b

Runtime for the Lanczos algorithm is proportional to ne, where n is the number of columns in the input matrix and e is the number of nonzero entries in the matrix. By removing these two columns, we only decrease n by 2; but reduce e by nearly 8%. This reduction in the number of nonzeros in the system is not quite reflected in the reduced time taken to solve the system via Lanczos for this example; however, non-optimal code (and possibly operating system overheads) may account for this discrepancy. For some smaller datasets, we did obtain savings in keeping with theoretical speedup (as much as 14% for a 6000×5000 system modulo a 20 digit prime). We also save a certain amount of space by removing this column, although this is rather small. For the first example above, we can remove 17,708 nonzeros. Assuming twelve bytes per nonzero (four for the position, four for the value, and four for a pointer), removing the generator column allows us to save some 200Kb of memory, which would be doubled if, as is often the case, the matrix is (rather wastefully) also held in core memory in its transposed form. Recall from chapter 3 that computing a root of 2 to use as a generator took very little time (less than 10 hundredths of a second for each example tried), and so the savings in linear algebra can outweigh the initial effort quite considerably.

Generator choice and SGE

Using the SGE→Lanczos model, we would hope to obtain similar gains. An added advantage – although of course one need not actually make use of it – is that removing the column corresponding to factor base element 2 will remove some of the largest coefficients in absolute value. This should postpone the effects of coefficient blowup and may allow us to run the SGE procedure for longer.

Matrix	Time to solve with -1 and g columns	Time to solve without -1 and g columns	Saving (%)
$22,710 \times 22,710$	265,126	255,390	3.67
$28,535 \times 23,777$	133,529	127,978	4.16
$34,290 \times 24,489$	86,268	82,992	3.80

Table 6.3: Time taken via SGE→Lanczos with/without g column - 30b

Table 6.3 shows timings for the 30 digit dataset, again having 0, 20% and finally 40% excess rows, as before. We see immediately that the effectiveness of removing the generator column is *not* so pronounced as when solving directly via Lanczos; although we are still saving time overall. The reason for this is that the savings are only really being applied to the small submatrix output from SGE. In the SGE routine itself, heavy columns were deactivated anyway. Since we then process a smaller (and rather denser) matrix with Lanczos, a lower proportion of the nonzeros are represented in the -1 and g columns than was originally the case. For the first system in table 6.3, we find that some 3.66% of the total weight of the submatrix occurs in the -1 and g columns, as opposed to around 8% in the original system. Savings will thus be reduced.
We found that we *could* reduce the matrix slightly further prior to numerical breakdown if the generator column was removed. However, this extra reduction was only a matter of one or two rows/columns smaller. As an example, we could reduce an input matrix of dimension $28,535 \times 23,777$ to a $1,112 \times 1,078$ matrix if we removed the generator column, but the best we could do if retaining the generator column was an output matrix of $1,115 \times 1,081$. Further, as mentioned, there is no benefit in going to these extremes. The solution time for this system with an output matrix of $3,138 \times 2,988$ was 1,280 seconds, while for the smallest possible output matrix, the solution time was 5,058 seconds - maximum reduction in SGE does not generally lead to an optimal solution time.

6.2.5 Effect of partial relations

We want to solve for as many of the factor base elements as possible, to maximise the size of our database of known values. Using a certain amount of denser relations will allow us to increase this coverage of values. We can then recover more values and improve the performance of the second phase of the method; but this compromises the efficiency of methods for solving sparse linear systems, as shown in table 6.4.
Here we used the SGE→Lanczos→back substitution method to solve a system of full relations (modulo a 40 digit p) in 17,985 unknowns, where the 10% excess rows were either further fulls, or else fulls derived from 1, 2, 3 or 4-partials. This increases the number of values in our solution vector, but takes its toll on the time taken to resolve

17,985 fulls +	Coverage	Loss	Nonzeros	Time to solve
1,798 fulls	17,066	5.11%	270,996	330,908
1,798 via 1P	17,194	4.40%	286,389	384,922
1,798 via 2P	17,282	3.91%	305,490	442,635
1,798 via 3P	17,435	3.06%	335,388	488,242
1,798 via 4P	17,737	1.38%	450,496	502,723

Table 6.4: Increased factor base coverage using fulls via partials - 40F

the system - we had hoped that the SGE routine would reject the small number of dense rows without taking too much more time, but that the increased coverage would allow us to recover more values. This, however, was not the case. We are better off concentrating on making the matrix as sparse as possible, as in factoring, and increasing coverage in a secondary step prior to computation, as discussed in the next chapter.

6.3 Summary

In this chapter we have described an implementation of structured Gaussian elimination primarily based on that described by Bender and Canfield [11]. We have shown that it is possible to reduce the size of a system by as much as 95% or more, and that equivalently high (as much as 98%) reductions in Lanczos complexity are possible. Overall times for solution of the system using a combination of SGE, Lanczos and back substitution can be completed in less than 5% of the time taken to solve the system using Lanczos alone. This code is currently being used as part of a Function Field Sieve implementation for fields of characteristic 3 (Granger et al. [50]).

We have endeavoured to clarify the operation of this procedure from an implementation point of view - one of the aims of this study was to provide documentation to be of assistance to those intending to carry out their own implementation. The idea of taking more rows in order to speed up the algorithm is mentioned by LaMacchia and Odlyzko [70] and is confirmed by the results of our implementation. It remains the case, however, that structured Gaussian elimination involves a certain amount of 'trial and error'. Results suggest that it is preferable to terminate the SGE algorithm prior to maximum reduction in order to reduce the workload for a secondary algorithm such as Lanczos, but the exact proportions vary. The generation of such datasets is of course rather time consuming, but papers such as [70] describing implementations of SGE on datasets of up to 10^5 or more unknowns show that the SGE algorithm can be used to reduce systems by an order of magnitude or more in a relatively short timescale; drastically speeding up the overall solution procedure.

We have shown that choosing the generator of $(\mathbb{Z}/p\mathbb{Z})^*$ such that it is a small factor base element can give noticeable savings in the linear algebra step, but that these

gains are not so pronounced when using the SGE→Lanczos model, compared to using Lanczos directly. Of course, we also need to change bases before we can say if time is saved overall. We also noted that even adding a small number of relations derived from large prime variant techniques has an immediate negative effect on the time taken to solve the linear system, emphasising the importance of techniques to reduce the weight of such relations: it is better to maintain sparseness than to try to represent as many factor base elements as possible.

Chapter 7

Discrete Logarithm Computation

Here we examine the impact of large prime variants on the final step of the index calculus method: actual computation of discrete logarithms. We look at the effect of reusing data from our large prime variant relations in order to speed up this final part of the computation procedure.

7.1 Overview

We first recall the details of phase three of the index calculus method, and note certain suggested improvements to the basic method which have been proposed. We subsequently discuss how we may increase the number of values for which we know the discrete logarithm, by reusing our sets of partial relations.

7.1.1 Final steps

Final discrete logarithm computation follows the same basic method as relation generation. Recall that to compute the discrete logarithm of x, we pick a random a and compute xg^a. If this number is B-smooth, then

$$xg^a \bmod p = \prod q_i^{e_i}, \quad q_i \in P$$

and so

$$DL(x) + a = \sum e_i DL(q_i)$$

and we can use our values for $DL(q_i)$ to compute $y = DL(x) \bmod p$.
As before, more sophisticated methods exist, for example those described by Coppersmith et al. [28]. Here a similar strategy to that of the Waterloo variant is used. We first represent $xg^a \bmod p$ as a product of medium sized primes, and subsequently use sieving techniques to compute the discrete logarithms of the medium sized primes using the discrete logarithms of the factor base elements.

7.1.2 Extending the factor base

Obviously, if we have not solved for some proportion of the factor base elements q_i we will have more difficulty finding a value a for which both $xg^a \bmod p$ is smooth *and* for which we hold the discrete logarithms of all the factor base elements used in the factorisation of this number. As we noted in the previous chapter, in general we will not solve for all values in the factor base. We may find that some elements were not actually represented in any relation; or we may 'lose' some values during the linear solve itself, for one reason or another.

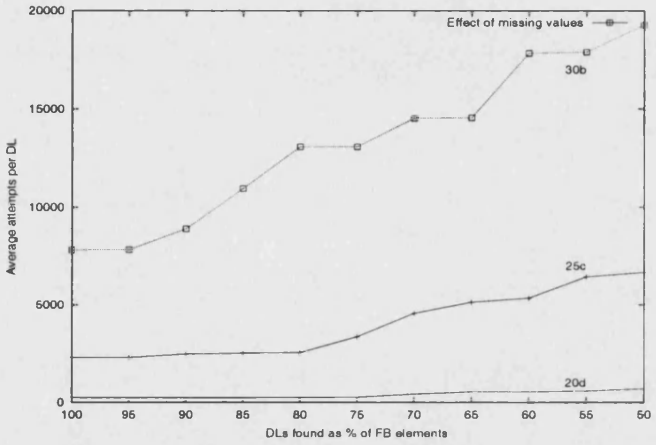

Figure 7-1: Average attempts needed to compute a given discrete logarithm

It would be of benefit to try to maximise the amount of 'factor base discrete logarithms' in order to reduce the number of attempts we need to compute a given discrete logarithm. The number of attempts we need in order to find a suitable a value and thus compute a discrete logarithm is illustrated in figure 7-1. Here we show the average number of attempts (per logarithm) it took to compute the discrete logarithms of 50 values chosen uniformly in $(\mathbb{Z}/p\mathbb{Z})^*$. The x axis represents the percentage of factor base elements for which we hold the discrete logarithm – 100% on the left, down to 50% on the right. As one would imagine, when we are missing half the factor base values, we take considerably more attempts to find a value a such that $g^a \bmod p$ factorises using only these remaining elements. The number of attempts grows in a rather stepped manner – this reflects when we have 'got lucky' in some way and happened upon a suitable value quickly. Notice, however, that a sudden increase in the number of attempts needed occurs sooner for the 25 digit dataset than for the 20 digit dataset, and similarly for the 30 digit dataset compared to the 25 digit dataset. Our smoothness bound B grows considerably slower than our modulus p across these datasets, and so the effect of missing values is felt that much sooner as p increases.

If we assume that we have done the best we can in our linear solve step, is there any

way to add to our database of known values? One simple extension could have been added to the relation building phase. Suppose we find a value a such that $g^a \bmod p$ is actually prime. We could store this a value as a 'direct' discrete logarithm, and add the prime value to our factor base. Our factor base would then increase dynamically as we build relations. In practice, however, this method is of no value. Unless one imposes a restriction on the size of this remainder, one is forced to carry out a primality test each time we choose a new a value. The increased load on the relation generation phase is unacceptable. On the other hand, if one imposes a maximum bound for such 'direct' discrete logarithms, the chances of actually happening on a prime value are negligible – for a 100 digit prime, if we set our bound at 10^{10} we would have a chance of some 1 in 10^{90} of hitting a prime within our range.

Another way in which we can try to maximise our number of known values can be used in final discrete logarithm computation itself. As we try to find a suitable number a such that $xg^a \bmod p$ is smooth, we can examine attempts where this is not the case. If we would have had a smooth value but for the fact that we do not hold the discrete logarithm for one of the prime factors, we can use this relation to compute the discrete logarithm of this element using the known values in our possession. In this way we can hopefully add to our database. It is unlikely that we would want to take the time to compute all missing values in the solution vector directly.

7.2 Using partial relations

We now investigate how we can use partial relations to improve phase three of the index calculus algorithm, firstly using our 1-partial relations, and subsequently considering the use of 2, 3 and 4-partials.

7.2.1 Results using 1-partials

We may add to our 'known' factor base values in a quite straightforward manner using our partial relations[1]. Once we have resolved our linear system and obtained the discrete logarithm of as many of our factor base elements as possible, we can try to extend our solution of known values by using the set of 1-partial relations. To do this we simply back substitute over the 1-partials to obtain the discrete logarithm of each large prime. We may then add the large prime to our factor base, and its discrete logarithm to our solution vector. In this way we should gain many more values and effectively make use of the 'single large prime' variant in the final stage of the index calculus method.

We can predict the number of extra values obtainable from such a technique via a

[1]Thomé [127] mentions the possibility of using 1-partials for such a tactic, but again this work was carried out independently. The idea is also briefly mentioned by Weber [133].

simple adaption of the estimate from Lenstra and Manasse [75] for the amount of fulls we can get from a set of 1-partials, which we now recall from the chapter 4.

Lemma 7.2.1 (Lenstra and Manasse [75], from lemma 3.1). *Let R be a set of partial relations and let $Q = q : q$ prime, $B_1 < q < B_2$. Assume that for each $u \in R$ we have probability*

$$P_q = q^{-\alpha} / \sum_{q \in Q} q^{-\alpha}$$

that $q = q(u)$ for some $q \in Q$. Then the number of matches amongst the large prime relations - i.e. the number of full relations we can obtain - is equal to

$$\#R - \#Q + \sum_{q \in Q} (1 - P_q)^{\#R}$$

Using this result we can thus deduce the number of 'large prime discrete logarithms' which we can obtain from R.

Corollary 7.2.1 (Yield of 1-partial back substitution). *The number of large prime values for which we can obtain the discrete logarithm, given a set R of 1-partial relations and the discrete logarithms of all factor base elements, is*

$$\#Q - \sum_{q \in Q} (1 - P_q)^{\#R}$$

Proof. If one considers the set of 1-partials as a graph by adding the vertex 1 as discussed in chapter 4, we have

$$\#\text{ fulls} = \#R + \#C - (\#V + 1) = \#R - \#V$$

where C is the set of components in the graph, and V is the set of unique large primes in the dataset. Since $\#C = 1$, if we hold all factor base element discrete logarithms then we can obtain $\#V$ large prime discrete logarithms, where

$$\#V = \#Q - \sum_{q \in Q} (1 - P_q)^{\#R}$$

□

As before, we can approximate $P_q \approx q^{-\alpha} / \sum_{p \in Q} p^{-\alpha}$ using a binomial expansion as shown by Boender and te Riele [16], and use the same value for α as we used to predict yield of resolving 1-partials in chapter 4. In what follows, we assume that we have solved for all factor base values in the linear algebra step.

Yield of back substitution over 1-partials

Table 7.1 shows both the estimated and actual number of further values we can find by back substituting over 1-partial relations for our three datasets. Here 'FB' signifies 'factor base', and 'extra values' indicates the additional values now known thanks to the 1-partials.

Dataset	FB elements	Number of 1-partials	Estimated extra values	Actual extra values
20d	6,058	197,742	189,219	188,790
25c	9,593	383,844	371,085	370,079
30b	25,998	768,435	720,253	719,261

Table 7.1: Extending factor base by back substitution over 1-partials

As before, we can examine how this affects our discrete logarithm computation. Figure 7-2 shows the number of attempts needed to compute the same 50 discrete logarithms as before. We see that the average number of attempts per logarithm has dropped to around a quarter of the number taken using the original factor base. Much as before, of course, we still suffer if values are missing, since we cannot recover so many 'large prime' discrete logarithms.

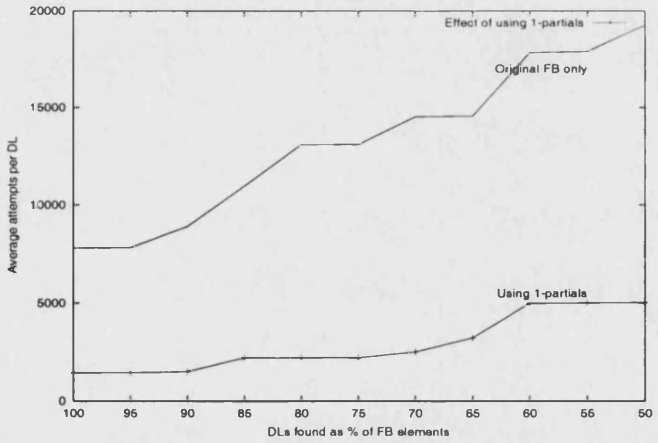

Figure 7-2: Average attempts needed to compute a given discrete logarithm using extended factor base – 30b

We note that we could also use large prime relations to complete values. When we have back substituted over 1-partials, we can possibly find further factor base values by back substituting over 'pruned' 1-partials (i.e. the set of 1-partials where the large primes occur more than once).

Time savings

The downside of extending the factor base is that the time taken for each attempt goes up. Here we are simply trial dividing with each element of our extended factor base. We can of course easily remedy this by re-using the 'large prime variant' technique we used in original relation generation. We would then proceed as follows: we first trial divide by all elements of the original factor base. We then consider the remainder. If it is less than the largest prime in our extended factor base (for which we hold the discrete logarithm) we can look up this value with, for example, a binary search, or via some kind of hash table lookup. In this manner, we will of course miss out on any values where the remainder consists of the product of more than one large prime value. However, it should turn out that the increased speed we gain by avoiding excessive trial division will more than compensate for this increase in the number of attempts we need to take. We will, in the very worst case, take the same number of attempts that we took in trial division using the original factor base only, and may expect that this 'early abort' strategy will significantly reduce the time of each attempt.

Dataset	Factor base elements	Attempts per DL	Time per attempt	Time per DL
20d	6,058	342	0.0481	16.48
20d	194,848	72	1.7978	129.44
25c	9,593	1,874	0.0808	151.48
25c	379,672	535	4.6761	2,504.24
30b	25,998	7,816	0.2274	1,777.55
30b	745,259	1,448	9.8288	14,235.05

Table 7.2: Average time per attempt via trial division with 1-partial extended factor base

Tables 7.2 and 7.3 show timings and the average number of attempts needed to compute a discrete logarithm for each dataset, firstly using trial division, and then using the large prime variant approach. As before, this is the average over 50 computations.

Dataset	Factor base elements	Attempts per DL	Time per attempt	Time per DL
20d	6,058	342	0.0481	16.48
20d	194,848	96	0.0445	4.30
25c	9,593	1,874	0.0808	151.48
25c	379,672	625	0.0758	47.44
30b	25,998	7,816	0.2274	1,777.55
30b	745,259	2,744	0.2179	598.07

Table 7.3: Average time per attempt via large prime variant with 1-partial extended factor base

Looking at results in both tables 7.2 and 7.3 we see that we do indeed take rather more attempts when using the large prime variant approach over a full trial division approach with our extended factor base of known values – for example, for dataset 30b we take 2,578 attempts using the large prime variant rather than only 1,448 when using trial division. However, the actual time taken is considerably less per attempt; with the overall effect being a net speedup. In fact, using the large prime variant allows computation of discrete logarithms in around 30% of the time it takes when using the original factor base only; and around 3% of the time it takes if extending the factor base but simply using trial division.

7.2.2 Results using 2-partials

Why stop there? We could, with a little more effort, carry out a similar operation using our 2-partial relations and extend the factor base further. This is of course not quite so trivial – one needs to track the values already found via a hash table, and potentially loop through the dataset a number of times in order to solve for as many values as possible.

Yield of back substitution over 2-partials

The number of extra values we can get by back substituting over both 1 and 2-partial relations is shown in table 7.4. Unsurprisingly, we are able to further extend our

Dataset	FB elements	Number of 1-partials	Number of 2-partials	Extended FB elements
20d	6,058	197,742	238,018	334,910
25c	9,593	383,844	788,239	885,875
30b	25,998	768,435	1,494,109	1,855,503

Table 7.4: Extending factor base by back substitution over 1 and 2-partials

database of known discrete logarithms quite considerably. Again, although we can reduce the number of attempts needed to find a suitable a value by using trial division (as shown in figure 7-3) the actual time needed to make each attempt negates this advantage as the factor base gets larger, as we now show.

Time savings

We again have the option of trial division or using the large prime variant as above. We also have, of course, the option of using the double large prime variant in the computation step. This, as when applied in the relation generation phase, requires an application of a factoring algorithm such as Pollard ρ when the remainder is both

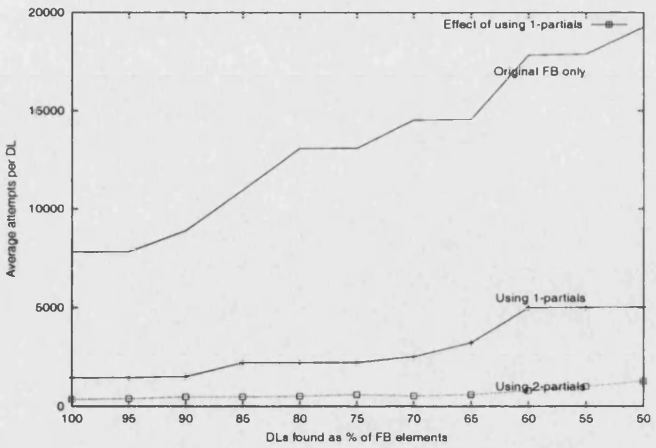

Figure 7-3: Average attempts needed to compute a given discrete logarithm using doubly extended factor base – 30b

composite and less than the square of the largest factor base element for which we hold the discrete logarithm. Results are shown in tables 7.5, 7.6 and 7.7.

Dataset	Factor base elements	Attempts per DL	Time per attempt	Time per DL
20d	6,058	342	0.0481	16.48
20d	334,910	36	2.9390	107.86
25c	9,593	1,874	0.0808	151.48
25c	885,875	133	10.4400	1390.40
30b	25,998	7,816	0.2274	1,777.55
30b	1,855,503	348	24.2549	8450.89

Table 7.5: Average time per attempt via trial division with 1 and 2-partial extended factor base

As mentioned, we see in table 7.5 that using trial division across this further extended factor base results in a dramatically increased time taken for each smoothness test. As we would expect, we take fewer attempts when we have a larger factor base but the time taken for each attempt becomes so large that any benefit in getting these extra values becomes obsolete.

In tables 7.6 and 7.7 we can compare the use of the single and double large prime variants on this further extended factor base. In table 7.6 we use the single large prime method on a factor base that has been extended by back substitution over both 1 and 2-partial relations. As before, we find that although with this method we take more attempts than by full trial division, the actual time for a given attempt is very much reduced, to the extent that we take less time per attempt than we did when using the large prime variant in the previous section. However, as shown in table 7.7, when using

Dataset	Factor base elements	Attempts per DL	Time per attempt	Time per DL
20d	6,058	342	0.0481	16.48
20d	334,910	63	0.0439	2.80
25c	9,593	1,874	0.0808	151.48
25c	885,875	366	0.0747	27.36
30b	25,998	7,816	0.2274	1,777.55
30b	1,855,503	1,391	0.2200	306.18

Table 7.6: Average time per attempt via large prime variant with 1 and 2-partial extended factor base

the double large prime variation we do not get a further speedup. In fact, the time taken increases considerably[2]. The number of attempts we take has actually dropped compared to the number we took when using the large prime variant; but the extra load on processing – a primality test, a factorisation with Pollard ρ plus a second lookup in the list of known values – means net performance is reduced.

Dataset	Factor base elements	Attempts per DL	Time per attempt	Time per DL
20d	6,058	342	0.0481	16.48
20d	334,910	44	5.0291	221.38
25c	9,593	1,874	0.0808	151.48
25c	885,875	146	8.9609	1,312.05
30b	25,998	7,816	0.2274	1,777.55
30b	1,855,503	406	1.2756	432.67

Table 7.7: Average time per attempt via double large prime variant with 1 and 2-partial extended factor base

It is best, then, to feed as many relations into back substitution as one can – back substitution is cheap, and a larger database of known discrete logarithms will always reduce the number of attempts one needs to take before one finds a smooth value when trying to compute an arbitrary discrete logarithm. This includes back substituting over the 1 and 2-partial relations to get discrete logarithms corresponding to as many large primes as possible. Further, it is of great benefit to re-use the 'large prime variant' strategy when actually carrying out the smoothness testing. For the dataset sizes we have considered here, it would not appear worthwhile to go further and use the double large prime variant in the final computation phase of the index calculus method, but for larger dataset sizes we would expect that the fewer attempts required by allowing two large primes would make such an approach more competitive.

[2]The 30 digit dataset does not follow the pattern set by the 20 and 25 digit datasets – this is due to it having a large prime bound which is less than the square of the original smoothness bound

7.2.3 Using 3 and 4-partials

When using a Waterloo-type approach, or indeed using higher order partial relations in general, one can of course try to recover still more values from these relations. Table 7.8 shows the number of values we can recover via back substitution over the 1 and 2-partials from the datasets used in chapter 5.

Dataset	FB elements	Number of 1-partials	Number of 2-partials	Extended FB elements
35A	4,024	166,925	1,564,183	910,069
40E	6,058	58,962	799,818	206,053
40Es	6,058	15,997	71,543	53,526

Table 7.8: Extending factor base by back substitution over Waterloo 1 and 2-partials

Going further, we can subsequently process the 3 and 4-partial relations. Table 7.8 shows the additional values we can gain. Totals are cumulative, that is to say, the column 'extended FB elements' refers to the total number of values for which we know the discrete logarithm up to this point. For dataset 40E, memory restrictions prevented us from processing all 3 and 4-partials. In table 7.9, the asterisks refer to the fact that the results obtained came from processing 10^6 3-partials and 10^6 4-partials. Dataset 40Es used smaller large prime bounds, and hence gives perhaps a better indication of practical yield. It is important to note that again we must take account of the exponents of the large primes if using Waterloo-derived data.

Dataset	Number of 3-partials	Extended FB elements	Number of 4-partials	Extended FB elements
35A	3,166,576	2,679,166	1,858,108	3,670,968
40E	3,508,419*	254,782	5,144,673*	272,552
40Es	147,603	134,488	142,551	212,485

Table 7.9: Extending factor base by back substitution over Waterloo 1,2,3 and 4-partials

Obviously, the more values we know, the faster our computation. We briefly note the effects of this further back substitution in table 7.10. Here we show the average time taken to compute a discrete logarithm for a 35 digit dataset. Based on our experiences above, we use the single large prime variant with table lookup against the values found by back substitution over the 1-partials (1P), 1 and 2-partials (1+2P), etc. For these larger datasets, we modified the computation routine such that it too takes a Waterloo approach on the value $xg^a \bmod p$ – the basic method becomes unfeasible for these sizes. We see again the power of extending the factor base of known values. We can bring the number of attempts needed to find a smooth value down to less than 0.5% of the number required using the basic factor base only. Since the time per attempt remains

Input	Factor base elements	Attempts per DL	Time per attempt	Time per DL
Original FB	4,204	50,085	0.0656	3,284.59
FB + 1P	160,525	1,794	0.0663	118.92
FB + 1,2P	910,069	643	0.0666	42.84
FB + 1,2,3P	2,679,166	291	0.0667	19.44
FB + 1,2,3,4P	3,670,968	222	0.0665	14.80

Table 7.10: Average time per attempt via Waterloo large prime variant with 1, 2, 3 and 4-partial extended factor base – 35A

more or less constant, we get an equivalent reduction in the time taken to compute a particular logarithm.

When considering datasets 40E and 40Es, we found that the double large prime variant gave improved performance over the single large prime variant – a speedup of between 22% and 65% for the examples here – when applied to dataset 40Es, but not when applied to dataset 40E. This is of course due to the smaller large prime bounds used in dataset 40Es, which restrict the number of primality tests and applications of Pollard ρ, and increase the chances of our holding the respective 'large prime discrete logarithms' in our extended factor base. This again underlines the advantages of using a smaller double large prime bound than simply B_1^3.

7.3 Cost of extending the factor base

We do, of course, also need to take account of the amount of work needed to actually compute these extra values for our factor base. Table 7.11 shows the time taken to back substitute over the original factor base (starting with back substitution over the original factor base using the solution vector from the Lanczos algorithm), followed by subsequent time taken to back substitute over the 1 and 2-partial relations for each dataset. From table 6.1, the fastest time taken to solve the 30 digit dataset using SGE and Lanczos was some 86,000 hundredths of a second.

Dataset	FB backsub time	1-partial backsub time	2-partial backsub time	Total time
20d	14	326	758	1,098
25c	29	731	2,433	3,193
30b	126	1,669	4,184	5,979

Table 7.11: Timings for back substitution over 1 and 2-partials

Back substitution over the unknown factor base values is very small in comparison. Back substituting over the 1-partial relations takes rather longer, due mainly to the

need to process more data. The time taken to process the 2-partial relations is again rather more pronounced, since we must pass through the dataset a number of times in order to maximise the yield. It is interesting to note that, in general, some 75% or more of the values obtained via 2-partials were found after the first pass through the dataset. This single pass took between 34% and 56% of the total time required to find all possible values (which required 9 passes through the set of 2-partials for the 30 digit dataset), as shown in table 7.12. It may then be beneficial to simply take a single pass through the data.

Dataset	Total DLs via 2-partials	Passes / Time	DLs found in single pass	Time	% DLs found / % time taken
20d	140,062	7 / 758.0	111,222	259	79 / 34
25c	506,203	8 / 2,433	373,068	989	74 / 41
30b	1,110,244	9 / 4,184	865,513	2,337	78 / 56

Table 7.12: Values found in single pass through partial data

Back substituting over the 3 and 4-partials is again more costly, as shown in tables 7.13 and 7.14. Since our Waterloo data comes from different datasets, we show the time for back substitution over factor base elements and 1 and 2-partials for these datasets, for comparison purposes. As before, we see a pronounced increase in going from back substitution over 1-partials only to back substituting over 2-partials, due to the need for repeated passes through the datasets.

Dataset	FB backsub time	1-partial backsub time	2-partial backsub time	Total time
35A	344	517	10,667	11,528
40E	1,643	214	9,009	10,866
40Es	1,882	58	577	2,517

Table 7.13: Timings for back substitution over Waterloo 1 and 2-partials

As a benchmark, the fastest times (in hundredths of a second, as usual) taken to solve each dataset via SGE and Lanczos was 20,295 for dataset 35A, 497,042 for dataset 40E and 530,835 for dataset 40Es - due the smaller large prime bound used, dataset 40Es has slightly longer cycles among the partial relations and as a result has a denser matrix than dataset 40E. In table 7.14, the final total column indicates total time for *all* back substitution.

Here, datasets 40E and 40Es differ only in the fact that dataset 40Es has a smaller double large prime bound than dataset 40E (it is $100{B_1}^2$ rather than B_1^3). This, as mentioned, brings the amount of data down quite considerably, and consequently speeds up the back substitution process. However, the number of large prime discrete

Dataset	3-partial backsub time	4-partial backsub time	Total time
35A	21,496	9,360	42,384
40E	8,624	9,542	29,032
40Es	1,478	1,001	4,996

Table 7.14: Timings for back substitution over Waterloo 3 and 4-partials

logarithms which we can obtain does not diminish commensurately. Again, we find

Input	Passes made	Values found	Time taken
1-partials	1	156,321	517
1-partials	1	156,321	517
2-partials	9	749,544	10,667
2-partials	1	531,133	2,900
3-partials	8	1,769,097	21,496
3-partials	1	1,293,761	7,490
4-partials	7	991,802	9,360
4-partials	1	901,777	5,622

Table 7.15: Values found in single pass through Waterloo partial data– 35A

that taking a single pass through the datasets gives us a large proportion of the total available values, but takes considerably less time – details for the 35 digit dataset are given in table 7.15. We found that the amount of time needed to find all possible values was not compensated by the associated speedup in computation – for the 35 digit dataset, for example, we saved a total of 25,511 hundredths of a second by taking a single back substitution pass across all partials. If using the single large prime variant on a factor base extended by 1,2,3 and 4-partials, we then take 493 hundredths of a second per logarithm, rather than 370 had we recovered all possible values in the back substitution step. We would thus need to compute 207 logarithms before the extra effort would become worthwhile. Overall, then, it is probably not worth taking the time to extract all possible information from the partial data unless one has to compute many logarithms.

Cost of choosing generator

We finally return to the idea of choosing the generator of the finite field under consideration. This allowed us to make certain savings in the linear algebra step of the index calculus method, but does require that we compute *two* discrete logarithms in order to change base. For the datasets under consideration here, dataset 40Es was generated using $\sqrt{2} \bmod p$ as a primitive root. Computation of this root using the

methods described in chapter 3 took 1 hundredth of a second. Dropping the 2-column in the linear algebra step saved us 24,156 hundredths of a second if using Lanczos only, but only 3,993 if using the SGE→Lanczos approach. Maximising the size of our factor base via back substitution over partials allowed us to compute a single logarithm in 4,291 hundredths of a second. Thus, for this example, choosing a new generator would only have been beneficial if solving directly via Lanczos or a similar iterative scheme without (or, perhaps, with less) preprocessing. Of course, if the published generator was itself 2, as may often be the case, we do not incur any costs from computing a new generator or changing bases in final computation, and can simply take advantage of the savings in time and space in the linear algebra step with no subsequent penalty.

7.4 Summary

In this section we illustrated the impact of an incomplete solution vector on phase three of the index calculus method. It was noted that, as datasets get larger, missing values have an immediate detrimental effect on performance.

We subsequently used 1-partial relations to extend the solution vector of known values by simple back substitution. This allowed us to use around one third as many attempts to compute a given discrete logarithm as it had taken us using the original solution vector. This underlines the advantage of using 1-partial relations for discrete logarithm computation – in addition to providing us with many full relations for little additional effort, they allow us to gain further useful values to add to our discrete logarithm database. This secondary advantage has no analogy in factoring. Testing for smoothness of course then becomes considerably slower if using full trial division, but using the large prime variant as in the relation generation step speeds this up considerably.

We investigated going a step further with this technique by additional back substitution over the 2, 3 and 4-partial relations. This is rather more involved than it was for the 1-partial relations due to the need for multiple passes through the data to recover more values. We found that, since most values are found after a single pass, it was better to terminate the back substitution procedure rather than trying to recover all possible values. For larger datasets, however, back substitution will be much cheaper in comparison to discrete logarithm computation, so it is likely that here one should extract the maximum number of values from the partial data. It may be possible to improve the efficiency of this step by reusing the graph structure determined when resolving the partial relations after the relation generation step.

The cost of back substitution is small compared to the time needed to solve the linear system at the end of the precomputation phase. For our tests, we found that it was fastest to use the single large prime variant in computation, coupled with a hash table lookup to recover the value of the large prime. As datasets get larger, the double large

prime variant would probably outperform this. Alternatively, Weber [134] suggests using the lattice sieve of Pollard [104], using the known large prime values as the 'special q'.

Finally, we showed that there can be an advantage to be gained by choosing a different generator for use with the index calculus method. However, this benefit is firstly small, and secondly only seems to outweigh other costs when used with direct application of an iterative scheme in the linear algebra step. On the other hand, it is not difficult to implement, and should probably be used if a scheme such as the Lanczos (or popular block Wiedemann) algorithm is used with limited preprocessing.

Part IV

Conclusions

Chapter 8

Conclusions and Further Work

8.1 Summary and results

We now recap on the contents of this thesis and review the results and conclusions obtained, on a chapter-by-chapter basis.

8.1.1 Part I

Chapter 1

In chapter 1 we gave a brief overview of cryptography as a subject, since it is in public key cryptographic applications that one finds the principal motivation for discrete logarithm computation. We discussed both symmetric and asymmetric schemes, and attempted to give an idea of the current state of the art. We subsequently gave a breakdown of the thesis layout.

Chapter 2

Chapter 2 provided a background to subsequent chapters concerning discrete logarithm computation. After defining the discrete logarithm problem and identifying its relevance to public key cryptography, we proceeded to examine several methods which have been proposed for discrete logarithm computation. The last of these, the index calculus method, was discussed in more detail and various possible improvements which have been proposed over the years were reviewed.

8.1.2 Part II

Chapter 3

Following an overview of the ideas behind the index calculus method, we considered the gains which can be made by choosing a different generator g' to that given as part of a public key. The complexity of the discrete logarithm problem is independent of

the generator of the finite field, so we can choose this generator to be as small a prime as possible. We have shown that, due to the elementary fact that $\log_g g = 1$, we can remove a certain amount of nonzero values from the ensuing matrix. The exact savings one can make can be as much as 7% of the total weight of the matrix, although this amount does decrease as both larger systems and more complex relation generation procedures are considered.

In order to maximise savings, we can take the generator g' to be either 2 or some k^{th} root of 2 modulo p. Such a value always exists, and, with the assumption that we know the prime factorisation of the group order $p-1$, such a value should be computable without a great deal of effort. Such a change does however then require computation of two discrete logarithms in order to change bases back to the original base g. In practice we found that, for the examples under consideration, the benefits in the linear algebra step could outweigh these extra costs if applying the Lanczos algorithm directly, but not if using preprocessing with structured Gaussian elimination. However, as datasets get larger, so will the submatrix output from SGE, and thus savings will hopefully outweigh the cost of changing bases.

The ideas and results of this chapter appear to be completely new, although the idea is of course a simple enough observation. Such a technique does not have a parallel in the application of the index calculus method to factoring problems.

Chapter 4

In this section we looked at speeding up the relation generation phase of the index calculus method for discrete logarithm computation by using large prime variants as applied to factoring by Lenstra and Manasse [75]. As noted, whilst this work was carried out independently, results of Thomé [127] and Weber [131] show that this idea had already been shown to be of benefit in discrete logarithm computation. However, details of the methods used are not discussed in any depth, and our hope in this work was to clarify the available techniques from a discrete logarithm viewpoint.

We highlighted the fact that the key difference between the use of these techniques for factoring and for discrete logarithm computation is the potential restrictions on the type of cycle found when using a graph theoretic approach in the latter case – at the most basic level, we need to look for even cycles.

We have shown that, in practice, the two large prime variant for discrete logarithm computation is within ε of being as effective as when applied to factoring. Loss is due to the need for either even cycles, or odd cycles including the special vertex 1. Hence, the addition of 1 to the graph is of greater importance for discrete logarithm computation than it is for factoring. However, we showed that, once all possible outcomes are identified, one may try to maximise the yield from a given dataset. We may join odd cycles together to form further even cycles, and can combine loops with 1-partials to

create further fulls.

We showed how one can easily identify even or odd cycles and order edges such that the large primes may be eliminated as the cycles are constructed. This is more efficient than building cycles and subsequently solving a linear system to eliminate the primes in each cycle, but without further information on how this technique was carried out we cannot say exactly. The only discussion of using two large primes for discrete logarithm computation that we could find in the open literature (Weber [131]) reports losses of some 2–3% in the number of cycles which can actually be resolved. For our tests, we found we could resolve almost all cycles found. The exceptions consisted of the few isolated loops which could not be matched to a 1-partial.

Processing both 1-partial and 2-partial relations together (i.e. adding the vertex 1 to the graph), in addition to allowing us to resolve most odd cycles, also gives both a drastic improvement on yield compared to processing each set separately, and a sparser set of full relations after processing, due to shorter cycle lengths. We may then either terminate sieving that much sooner if one had an indication of potential yield from a given set of partial relations, or else build a large number of excess relations which we may then use in a back substitution step in an effort to recover the discrete logarithms of a greater number of factor base elements.

The price of the success of processing 1-partials and 2-partials together is the increased memory requirement and processing time used by the graph algorithms. However, we have also illustrated the effect of reducing the bound used for the 2 large prime relations – one can reduce this quite considerably before seeing a proportionate drop in the number of full relations one can still resolve.

Chapter 5

In chapter 5 we built on the ideas of chapter 4 and investigated the effect of using large prime variants with the 'Waterloo variant' of the index calculus procedure. This allows use of up to four large primes for a rational factor base. The key difference between this work and the use of '2 + 2' large primes by Dodson and Lenstra [38] for factoring and Weber [131] for discrete logarithm is that here all large primes are processed together. In this sense the work is more closely related to that of Leyland et al. [81], where three large primes are used.

We first showed how use of the Waterloo variant affects the methods of chapter 4. We showed that a similar yield from partial relations is possible, but that resolving cycles is slightly more complex due to the differing exponents of the large primes in relations generated by the Waterloo variant. We must be rather more careful with the order in which we process edges in a cycle, in order to correctly eliminate all large primes. Again, such problems do not occur in factoring applications, since there we simply need each vertex to occur an even number of times.

We subsequently moved on to consider 3 and 4-partial relations. We initially showed how one can determine full relations in a relatively straightforward manner if a (hyper)cycle contains at most one 3 or 4-partial, using the assumption that the vertex 1 is the root vertex of the major graph component. Going further with this means of resolving cycles, we discussed how the method of Leyland et al. [81] – essentially the same as that of Dodson and Lenstra [38] – could be adapted for the discrete logarithm case, such that all hypercycles *built* by the procedure could be resolved for the discrete logarithm case. It remains the case that certain situations (notably loops), while being straightforward to resolve for factoring purposes, cannot be resolved for the discrete logarithm case. While these techniques will be successful for n-partial relations, however, they will create rather longer cycles than certain other methods.

8.1.3 Part III

Chapter 6

In this chapter we took a closer look at the linear algebra step of the index calculus method. We described an implementation of structured Gaussian elimination (SGE), and highlighted various practical difficulties which may arise. We then described the results of our implementation in conjunction with the Lanczos algorithm, and obtained results similar to those described by LaMacchia and Odlyzko [70]. As in [70], we found that using excess rows can speed up the overall solution time. The code implemented during this study is currently being used in an attempt to compute discrete logarithms of some 350 bits via the function field sieve in a finite field of characteristic 3 (Granger et al. [50]). In initial testing, a system of 10^5 equations in 10^5 unknowns could be solved in under 5 hours.

We then observed the practical effect of removing the generator column as described in chapter 3. This resulted in time savings both when applying the Lanczos algorithm alone or coupled with SGE preprocessing, although time savings were not so pronounced in this latter case. We had hoped that removal of the generator column would also have the added benefit of delaying the effect of coefficient blowup in the structured Gaussian elimination routine such that we could output a smaller matrix, but, while this was the case, the effect was negligible and also occurred well after the optimal stop point for SGE for each dataset tested.

Finally, we noted the key issue of factor base coverage in the linear algebra step. This has no analogy in factoring, and suggests that, while we wish to make the linear algebra step as efficient as possible, we also want to maximise the number of solution values we can obtain. This issue does not appear to have been considered previously in the open literature. To this end, taking excess rows is again of benefit, such that further factor base values are represented in the matrix. However, we found that even a small amount of dense relations – those coming from resolving n-partials – had an immediate

detrimental effect on performance. As in factoring, then, it is best to concentrate on making the linear solve as efficient as possible by keeping weight to a minimum, and look at coverage in a secondary step.

Chapter 7

In this chapter we highlighted the effect of an incomplete solution vector on discrete logarithm computation. In general, we do not represent all factor base elements in the matrix, and so hamper our attempts to find a smooth value for which we know all factor base discrete logarithms in our our final computation step.
To remedy this, we made use of our partial relations in a back substitution step. This allowed us to greatly extend our database of known values, and led to considerable speedups in computation. Again, this has no analogy in the factoring application of large prime variant techniques. Using the single large prime variant in computation, with a table lookup on the large prime value, was found to outperform the double large prime variant for the datasets tested, although we would expect this to change for larger datasets.
We finally investigated further back substitution over all n-partials. This is not so straightforward as for 1-partials only, due to the need for repeated passes through large datasets. For the sizes considered here, we would recommend taking only a single pass through the datasets, which for our implementations managed to recover some 75% of the total values which we could hope to obtain, whilst taking, in general, less than the half the time. Using these methods, we found that our choice of generator could indeed lead to overall savings in time, but only when using an iterative scheme directly. However, if a change of bases were not necessary, then taking advantage of $g = 2$, if possible, will always lead to savings in the linear algebra step.

8.1.4 Conclusions

This thesis has tried to give an in-depth examination of the use of large prime variants for discrete logarithm computation modulo a prime p. Index calculus-type methods have been applied to both factoring and discrete logarithm computation with great success; however, it is often found that analogous applications are rather more complex in the latter case. We have demonstrated that the use of large prime variant techniques also follows this pattern. That said, we have shown that, in practice, such techniques can be brought to bear with (almost) an equal degree of success to that of their factoring cousins, such that in practice they can be applied to computation modulo a k-bit prime with the same effectiveness as in factoring a k-bit number. Indeed, large prime variant techniques can even be thought to be *more* effective in the discrete logarithm case, thanks to their potential application in the final phase of the method. The linear algebra step of the index calculus method is traditionally a more serious practical bottleneck for

discrete logarithm computation than it is for similar factoring methods. However, by considering the initial parameters for the method, and by reusing large prime variant data in final computation, we have shown that linear algebra can nonetheless redeem itself at other points in discrete logarithm computation.

8.2 Further work

The discrete logarithm problem has now been a topic for cryptography related research for some 30 years. In that time, some huge advances have been made, but many questions also remain unresolved. As noted by Odlyzko [100], for example, in the last 25 years, no substantial progress has been made for a general cyclic group with no structure. In this final section, we note some possible suggestions for future work, mainly pertaining to the results of this thesis.

The use of two or more large prime variants for factoring has led to impressive practical speedup. The fact that these gains also apply to the discrete logarithm case was demonstrated by Thomé and Weber, and has been underlined by the work of this thesis. It remains the case, however, that the examples given in this thesis are rather small – it would be useful to verify the practicalities of these techniques for larger datasets, although our results indicate that it is safe to assume that results in factoring implementations will carry across directly to the discrete logarithm case. A recent paper by Gaudry [44] gives an index calculus-type method for computing discrete logarithms on hyperelliptic curves. It would be interesting to investigate whether or not large prime variant techniques would adapt to this new situation.

We have shown how we can resolve relations in n large primes, so long as we have some set of relations involving a single large prime to give us a 'toehold'. It seems unlikely that a means of generating data involving many more than 3 large primes directly (i.e. per smoothness test) would be practical, but this may not be the case. Alternatively, one could go further with the 2×2 large prime approach used here to generate n-partials with $n > 4$, in a similar manner to the work of Cavallar [21]. At present, it is not easy to estimate the yield, given a set of n-partials with $n > 1$, without determining, for example, the structure of the hypergraph they would create. If using many processors in different locations, this would mean examining all relations found after a given amount of time at some central location – it would be much simpler to be able to give an estimate for yield based solely on the number of partial relations found and the smoothness bounds used. Extensions to the 1-partial estimate formula of Lenstra and Manasse [75] would be of practical benefit in order to simplify relation generation time estimates.

The question of density is very important for practical efficiency in the linear algebra step. It could be interesting to adapt the techniques of Denny and Müller [35] to the

hypercycle building method of Dodson and Lenstra [38] and Leyland et al. [81] with a view to reducing the length of cycles prior to resolving. To a large extent this is exactly the aim of Cavallar [20], although as this is directed towards factoring, the important (discrete logarithm) issue of coverage in the solution vector is not considered.

In the computation step, we have only examined the most basic techniques. Extending the factor base should give useful speedup to other methods of computation, such as that of Coppersmith et al. [28]. Other techniques such as lattice sieving while making use of the known large prime discrete logarithms should give improved performance. Further investigation would be required to verify this. Again, approximations of the number of unique large primes in some given number of n-partial relations would be useful to get an idea of the yield of back substitution. It was straightforward to adapt the method of Lenstra and Manasse [75] in the 1-partial case, but for n-partials with $n > 1$ we have a slightly different situation, in that 'singletons' can now also be of use – we essentially need to estimate the number of vertices in the component containing the special vertex 1; which will generally be the largest connected component of the (hyper)graph. It may be that the theory of random graphs could be used to provide such an estimate, with a certain bias given to the often-occurring vertex 1. Again, this requires further investigation.

Appendix A

History of Large Prime Variant Index Calculus

This section gives a brief summary of the history of using large primes with the index calculus method both for factoring and for discrete logarithm computation, with a view to providing a comprehensive list of references for those interested in such techniques.

A.1 Single large prime

The large prime variant appears to go back at least as far as the development of the Continued Fraction factoring method of Morrison and Brillhart [92], and is analysed from a factoring viewpoint by Pomerance [107]. It is described, as applied to discrete logarithm computation, in Odlyzko's survey paper [98] of 1984, and is used in most implementations described in papers from this point onward, both for factoring and discrete logarithm computation. See Boender [15] for a discussion of the technique, and its yield as applied to the Multiple Polynomial Quadratic Sieve for factoring. Estimates for the number of 1-partials one may expect to find for a particular set of bounds are given by Bach and Peralta [9], while estimates of yield from these 1-partials can be found in Lenstra and Manasse [75] and Morain [91].

A.2 Two large primes

We must now differentiate slightly: two large primes have been used in essentially two different ways. The classic paper in the field is that of Lenstra and Manasse [75] which applies to factoring and was originally presented at EuroCrypt in 1990; although apparently the notion of using two large primes goes back to ideas of Montgomery and Silverman in the mid 1980s[1]. The method described by [75] uses two large primes with

[1] According to a brief discussion on `sci.crypt`.

a *single* factor base – this approach was also used by Atkins et al. [8] to break the original RSA challenge message.

The possibilities when allowing at most two large primes then increase when one considers the 'Waterloo variant' approach of Blake et al. [13], which, for the purposes of this discussion, leads to checking two values for smoothness. In a similar manner, developments such as that of the Number Field Sieve (see Lenstra and Lenstra, Jr. (eds.) [73]), starting around 1988, permit the use of *two* factor bases, which again may be thought of as checking two values for smoothness. In these cases, for a maximum of two large primes per relation, we can either use the single large prime variant in the factorisation of two values, or apply the 'classic' two large prime variant to one value only.

Obviously, for both of these cases we have relations involving two large primes, but they differ both in the generation and resolution of partial relations. For the first case [75], we introduce additional work in relation generation due to the need to factor a composite value in order to identify our two large primes. However, when trying to eliminate the large primes, one may process all these relations together, since there is a single set of large primes which occur in all relations. For the second case, the situation differs in that we remove the need for an expensive 'splitting out' of the large primes; but we have added complications in resolving the partial relations, as noted by [73]. This is due to the fact that we cannot in general match a large prime occurring when factorising over one factor base with the same value should it occur when factorising over the second factor base. If using a graph-theoretic approach to resolve these relations, the graph of 2-partial relations is then bipartite. We note that this complication does not occur when using the Waterloo variant with a single factor base. A 'hybrid' approach is, of course, possible – Thomé [127] allows a maximum of two large primes per relations, but allows these to come from either '(1×2)' or '(2×1)' approaches, depending on the result of the first smoothness test. For an experimental investigation of the technique, see Boender and te Riele [16] (again a factoring paper). Semi-smoothness estimates are extended to the case of two large primes by Lambert [71].

A.3 Using n large primes

Use of more large primes essentially follows the same pattern – an n-large prime variant can be thought of as an $(x \times y)$-variant where $n = x \times y$. Here x is actually the number of values being tested for smoothness; and so is actually better represented as $x_1 \times x_2$ where x_1 is the number of factor bases being employed, and x_2 denotes the number of smoothness tests being carried out per factor base. x_1 will generally be one or two; x_2 will usually be one, but may be two (as is the case if using a 'Waterloo variant'). The

value y is the number of large primes we allow in each smoothness test. This definition is slightly restrictive in that one does not have to use the same number of large primes for each smoothness test – we give the above definitions simply to illustrate different situations.

A three large prime version of GNFS was implemented by Buchmann et al. [19]. Effectively a '$(1 \times 1) + (1 \times 2)$' variant, this allowed one large prime on the rational factor base and two on the algebraic factor base. A very different '(1×3)' approach was given by Leyland et al. [81] in 2002, where one allows three large primes in a single smoothness test using MPQS. A similar investigation using NFS is given by Cavallar [21], who also generalises semi-smoothness probability estimates to allow n large primes. This implementation allows three large primes on one factor base and two on the other; and so in our terminology can be considered a five large prime variant. This essentially extends the results of Dodson and Lenstra [38], who implemented a 2×2 approach as applied to NFS, and showed unpredictable yet explosive growth in the yield of the technique. Further demonstration of the power of using more large primes was given by Cowie et al. [30] in factoring RSA-130. The technique was subsequently adapted for discrete logarithm computation by Weber [131] using an NFS approach, although few details concerning the resolution of partial relations are given.

Appendix B

Dataset Details

Here we give a brief description of the main datasets used throughout this thesis. These were used to compute discrete logarithms modulo primes of between 20 and 40 digits. Using the basic method we computed discrete logarithms modulo primes of 20, 25 and 30 digits, using parameters as shown in table B.1. Datasets are tagged with a number corresponding to the digits in the prime modulus – thus, for example, 'dataset 30b' refers to a dataset used to compute logarithms modulo a prime of 30 digits. Each prime p was chosen such that $p - 1 = 2q$ for prime q. In general, we maximised the large prime bounds, but for dataset 30b the large prime bound was chosen such that large prime values would fit in a C++ 'long' datatype. Generators are simply the smallest prime to generate $(\mathbb{Z}/p\mathbb{Z})^*$.

Dataset	p	g	B_1	FB elements	B_2	B_3
20d	$10^{20} + 763$	2	60,000	6,058	B_1^2	B_1^3
25c	$10^{25} + 1879$	23	100,000	9,593	B_1^2	B_1^3
30b	$10^{30} + 1783$	5	300,000	25,998	2^{31}	$B_1 * B_2$
35A	$10^{30} + 3043$	2	40,000	4,024	B_1^2	B_1^3
40d	$10^{40} + 17407$	$\sqrt{2} \bmod p$	1.4×10^6	107,127	B_1^2	B_1^3
40E	$10^{40} + 17407$	$\sqrt{2} \bmod p$	60,000	6,058	B_1^2	B_1^3
40Es	$10^{40} + 17407$	$\sqrt{2} \bmod p$	60,000	6,058	B_1^2	$100B_2$
40F	$10^{40} + 17407$	$\sqrt{2} \bmod p$	200,000	17,985	B_1^2	B_1^3

Table B.1: Parameters used in generation of data

Using the Waterloo variant index calculus method we computed discrete logarithms modulo primes of 20, 25 and 30 digits using the same parameters as above, and subsequently computed modulo primes of 35 and 40 digits. For datasets 40E and 40Es we used the ideas of chapter 3 to choose a root of 2 modulo p as a generator for $(\mathbb{Z}/p\mathbb{Z})^*$. Dataset 40Es differs from dataset 40E only in that it uses a smaller 'double large prime' bound in order to restrict the amount of data collected.

Bibliography

[1] Adleman, L. M. (1979). A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography. In *Proc. 20th IEEE Symposium on the Foundations of Computer Science*, pp. 55–60.

[2] Adleman, L. M. and J. DeMarrais (1993). A Subexponential Algorithm for Discrete Logarithms over All Finite Fields. *Mathematics of Computation 61*, pp. 1–15.

[3] Adleman, L. M., K. Manders, and G. Miller (1977). On Taking Roots in Finite Fields. In *Proc. 18th IEEE Symposium on the Foundations of Computer Science*, pp. 175–178.

[4] Agrawal, M., N. Kayal, and N. Saxena (2003). PRIMES is in P. Preprint, v3. Available at `http://www.cse.iitk.ac.in/news/primality_v3.pdf`.

[5] Alladi, K. (1987). An Erdös-Kac Theorem for Integers Without Large Prime Factors. *Acta Arithmetica 49*, pp. 81–105.

[6] Anderson, R. (2001). *Security Engineering - A Guide to Building Dependable Distributed Systems* (1st ed.). New York, NY: John Wiley & Sons, Inc.

[7] Anderson, R. and R. Needham (1995). Robustness Principles for Public Key Protocols. In *Proc. of Crypto 95*, Volume 963 of *LNCS*, pp. 236–247. Springer-Verlag.

[8] Atkins, D., M. Graff, A. K. Lenstra, and P. Leyland (1995). The Magic Words are Squeamish Ossifrage. In *Proc. of AsiaCrypt 94*, Volume 917 of *LNCS*, pp. 263–277. Springer-Verlag.

[9] Bach, E. and R. Peralta (1996). Asymptotic Semismoothness Probabilities. *Mathematics of Computation 65*, pp. 1701–1715.

[10] Bach, E. and J. Shallit (1996). *Algorithmic Number Theory - Efficient Algorithms* (1st ed.). Cambridge, MA: MIT Press.

[11] Bender, E. A. and E. R. Canfield (1999). An Approximate Probabilistic Model for Structured Gaussian Elimination. *Journal of Algorithms 31*, pp. 271–290.

[12] Bernstein, D. J. (2000). How to Find Small Factors of Integers. *Mathematics of Computation* (to appear). Preprint available at `http://cr.yp.to/factorization.html`.

[13] Blake, I. F., R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone (1984). Computing Logarithms in Finite Fields of Characteristic Two. *SIAM Journal of Discrete Methods 5*, pp. 276–285.

[14] Blake, I. F., R. C. Mullin, and S. A. Vanstone (1984). Computing Discrete Logarithms in GF(2^n). In *Proc. Crypto 1984*, Volume 196 of *LNCS*, pp. 73–82. Springer-Verlag.

[15] Boender, H. (1996). The Number of Relations in the Quadratic Sieve Algorithm. Technical Report NM-R9622, CWI, Amsterdam. Available at `http://www.cwi.nl/ftp/CWIreports/NW/NM-R9622.pdf`.

[16] Boender, H. and H. J. J. te Riele (1995). Factoring Integers with Large Prime Variations of the Quadratic Sieve. Technical Report NM-R9513, CWI, Amsterdam. Available at `http://www.cwi.nl/ftp/CWIreports/NW/NM-R9513.pdf`.

[17] Boneh, D. and R. Venkatesan (1996). Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes. In *Proc. Crypto 1996*, Volume 1109 of *LNCS*, pp. 129–142. Springer-Verlag.

[18] Buchmann, J., M. J. Jacobson, Jr., and E. Teske (1997). On Some Computational Problems in Finite Abelian Groups. *Mathematics of Computation 66*, pp. 1663 1687.

[19] Buchmann, J., J. Loho, and J. Zayer (1994). An Implementation of the General Number Field Sieve. In *Proc. Crypto 93*, Volume 773 of *LNCS*, pp. 159–165. Springer-Verlag.

[20] Cavallar, S. (2000). Strategies in Filtering in the Number Field Sieve. Technical Report MAS-R0012, CWI, Amsterdam.
Available at `http://www.cwi.nl/ftp/CWIreports/MAS/MAS-R0012.pdf`.

[21] Cavallar, S. (2002). The Three Large Prime Variant of the Number Field Sieve. Technical Report MAS-R0219, CWI, Amsterdam.
Available at `http://www.cwi.nl/ftp/CWIreports/MAS/MAS-R0219.pdf`.

[22] Cavallar, S. et al. (2000). Factorization of a 512-Bit RSA Modulus. In *Proc. Eurocrypt 2000*, Volume 1807 of *LNCS*, pp. 1–18. Springer-Verlag.

[23] Clark, D. W. and L.-J. Weng (1994). Maximal and Near-Maximal Shift Register Sequences: Efficient Event Counters and Easy Discrete Logarithms. *IEEE Transactions on Computers 43*(5), pp. 560–568.

[24] Cocks, C. (2001). An Identity Based Encryption Scheme Based on Quadratic Residues. In *Proc. 8th IMA Cryptography and Coding*, Volume 2260 of *LNCS*, pp. 360–363. Springer-Verlag.

[25] Contini, S. (1997). Factoring Integers with the Self Initialising Quadratic Sieve. Master's thesis, University of Georgia. Available at `http://www.crypto-world.com/Contini.html`.

[26] Coppersmith, D. (1984). Fast Evaluation of Logarithms in Fields of Characteristic Two. *IEEE Transactions on Information Theory IT-30*, pp. 587–593.

[27] Coppersmith, D. (1994). Solving Homogeneous Linear Equations Over GF(2) via Block Wiedemann Algorithm. *Mathematics of Computation 62*, pp. 333–350.

[28] Coppersmith, D., A. M. Odlyzko, and R. Schroeppel (1986). Discrete Logarithms in GF(p). *Algorithmica 1*, pp. 1–15.

[29] Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2001). *Introduction to Algorithms* (2nd ed.). Cambridge, MA: MIT Press.

[30] Cowie, J. et al. (1996). A World Wide Number Field Sieve Factoring Record: On to 512 Bits. In *Proc. AsiaCrypt 1996*, Volume 1163 of *LNCS*, pp. 382–394. Springer-Verlag.

[31] Crandall, R. and C. Pomerance (2001). *Prime Numbers: a Computational Perspective* (1st ed.). New York, NY: Springer Verlag.

[32] Crouch, P. A. and J. H. Davenport (2001). Lattice Attacks on RSA-Encrypted IP and TCP. In *Proc. 8th IMA Cryptography and Coding*, Volume 2260 of *LNCS*, pp. 329–338. Springer-Verlag.

[33] Davenport, J. H. (1992). Primality Testing Revisited. In *Proc. ISAAC 1992*, pp. 123–129. ACM.

[34] de Bruijn, N. G. (1951). On the Number of Positive Integers $\leq x$ and Free of Prime Factors $> y$. *Nederl. Acad. Wetensch. Proc. 54*, pp. 50–60.

[35] Denny, T. and V. Müller (1996). On the Reduction of Composed Relations from the Number Field Sieve. In *Proc. ANTS II*, Volume 1122 of *LNCS*, pp. 75–90. Springer-Verlag.

[36] Diffie, W. (2001). Invited talk. *8th IMA Cryptography and Coding*, Cirencester, UK.

[37] Diffie, W. and M. E. Hellman (1976). New Directions in Cryptography. *IEEE Transactions on Information Theory IT-22*, pp. 644–654.

[38] Dodson, B. and A. K. Lenstra (1995). NFS with Four Large Primes: An Explosive Experiment. In *Proc. Crypto 1995*, Volume 963 of *LNCS*, pp. 372 385. Springer-Verlag.

[39] ElGamal, T. (1985a). A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory IT-31*, pp. 469-472.

[40] ElGamal, T. (1985b). A Subexponential-Time Algorithm for Computing Discrete Logarithms over GF(p^2). *IEEE Transactions on Information Theory IT-31*, pp. 473–481.

[41] Ferguson, N. and B. Schneier (2003). *Practical Cryptography* (1st ed.). New York, NY: John Wiley & Sons, Inc.

[42] Fouché Gaines, H. (1956). *A Study of Ciphers and their Solution* (1st ed.). New York, NY: Dover.

[43] Garrett, P. (2001). *Making, Breaking Codes* (1st ed.). London: Prentice-Hall.

[44] Gaudry, P. (2000). An Algorithm for Solving the Discrete Logarithm Problem on Hyperelliptic Curves. In *Proc. EuroCrypt 2000*, Volume 1807 of *LNCS*, pp. 19-34. Springer-Verlag.

[45] Gauss, C. F. (1986). *Disquisitiones Arithmeticae* (9th ed.). New York, NY: Springer-Verlag.

[46] GCHQ. (Government Communications Headquarters) `http://www.gchq.gov.uk`.

[47] Girault, M. (1990). An Identity-Based Identification Scheme Based on Discrete Logarithms Modulo a Composite Number. In *Proc. EuroCrypt 1990*, Volume 473 of *LNCS*, pp. 481–486. Springer-Verlag.

[48] Gordon, D. M. (1993). Discrete Logarithms in GF(p) using the Number Field Sieve. *SIAM Journal on Discrete Mathematics 6*, pp. 124–138.

[49] Gordon, D. M. and K. S. McCurley (1993). Massively Parallel Computation of Discrete Logarithms. In *Proc. Crypto 1992*, Volume 740 of *LNCS*, pp. 312 323. Springer-Verlag.

[50] Granger, R., A. J. Holt, D. Page, N. P. Smart, and F. Vercauteren (2003). Function Field Sieve in Characteristic Three. Preprint, University of Bristol.

[51] Hafner, J. and K. S. McCurley (1989). A Rigorous Subexponential Algorithm for Computation of Class Groups. *Journal of the American Mathematical Society 2*, pp. 837–850.

[52] Hardy, G. H. and E. M. Wright (2000). *An Introduction to the Theory of Numbers* (5th ed.). Oxford, UK: Oxford University Press.

[53] Hellman, M. E. and J. M. Reyneri (1982). Fast Computation of Discrete Logarithms in GF(q). In *Proc. Crypto 1982*, pp. 3–13. Plenum Publishing.

[54] Hestenes, M. R. and E. Stiefel (1952). Methods of Conjugate Gradient for Solving Linear Systems. *Journal of Research of the National Bureau of Standards 49*, pp. 409–436.

[55] Hildebrand, A. (1987). On the Number of Factors of Integers Without Large Prime Divisors. *Journal of Number Theory 25*, pp. 81–106.

[56] Hildebrand, A. and G. Tenenbaum (1993). Integers Without Large Prime Factors. *Journal de Théorie de Nombres de Bordeaux 5*, pp. 411–484.

[57] Holt, A. J. (2002). Sparse Linear Systems in Cryptography. In *Proc. 8th Rhine Workshop on Computer Algebra*, pp. 85–99.

[58] Holt, A. J. and J. H. Davenport (2003). Resolving Large Prime(s) Variants for Discrete Logarithm Computation. In *Proc. 9th IMA Cryptography and Coding* (to appear), Volume 2898 of *LNCS*. Springer-Verlag.

[59] Huizing, R. M. (1995). An Implementation of the Number Field Sieve. Technical Report NM-R9511, CWI, Amsterdam. Available at `http://www.cwi.nl/ftp/CWIreports/NW/NM-R9511.pdf`.

[60] Jenkins, B. (1996). A Hash Function for Hash Table Lookup. See `http://burtleburtle.net/bob/hash/doobs.html`.

[61] Joux, A. and R. Lercier (2000). Improvements to the General Number Field Sieve for Discrete Logarithms in Prime Fields. *Mathematics of Computation* (To appear).

[62] Joux, A. and R. Lercier (2002). The Function Field Sieve is Quite Special. In *Proc. ANTS-V*, Volume 2369 of *LNCS*, pp. 431–445. Springer-Verlag.

[63] Joux, A. and K. Nguyen (2001). Separating Decision Diffie-Hellman from Diffie-Hellman in Cryptographic Groups. Available at `http://citeseer.nj.nec.com/joux01separating.html`.

[64] Kahn, D. (1996). *The Codebreakers* (2nd ed.). New York, NY: Macmillan.

[65] Kerckhoffs, A. (1883, January). La Cryptographie Militaire. *Journal des Sciences Militaires IX*, 5–38.

[66] Knuth, D. E. (1981). *The Art of Computer Programming (vol 2)* (2nd ed.). Phillipines: Addison-Wesley.

[67] Koblitz, N. (1987). *A Course in Number Theory and Cryptography* (1st ed.). New York, NY: Springer Verlag.

[68] Kraitchik, M. (1922). *Théorie des Nombres*, Volume 1. Paris, France: Gauthier Villars et Cie.

[69] LaMacchia, B. A. and A. M. Odlyzko (1991a). Computation of Discrete Logarithms in Prime Fields. In *Proc. Crypto 1990*, Volume 537 of *LNCS*, pp. 616–618. Springer-Verlag.

[70] LaMacchia, B. A. and A. M. Odlyzko (1991b). Solving Large Sparse Linear Systems Over Finite Fields. In *Proc. Crypto 1990*, Volume 537 of *LNCS*, pp. 109–133. Springer-Verlag.

[71] Lambert, R. (1996). *Computational Aspects of Discrete Logarithms*. Ph. D. thesis, University of Waterloo. Available at `http://www.cacr.math.uwaterloo.ca/techreports/2000/lambert-thesis.ps`.

[72] Lanczos, C. (1952). Solution of Systems of Linear Equations by Minimized Iterations. *Journal of Research of the National Bureau of Standards 49*, pp. 33–53.

[73] Lenstra, A. K. and H. W. Lenstra, Jr. (eds.) (1993). *The Development of the Number Field Sieve* (1st ed.). London, UK: Springer Verlag.

[74] Lenstra, A. K. and M. S. Manasse (1989). Factoring by Electronic Mail. In *Proc. EuroCrypt 1989*, Volume 434 of *LNCS*, pp. 355–371. Springer-Verlag.

[75] Lenstra, A. K. and M. S. Manasse (1990). Factoring With Two Large Primes. In *Proc. EuroCrypt 1990*, Volume 473 of *LNCS*, pp. 72–82. Springer-Verlag.

[76] Lenstra, A. K. and A. Shamir (2000). Analysis and Optimization of the TWINKLE Factoring Device. In *Proc. Eurocrypt 2000*, Volume 1807 of *LNCS*. pp. 35–52. Springer-Verlag.

[77] Lenstra, A. K. and E. R. Verheul (2001). Selecting Cryptographic Key Sizes. *Journal of Cryptology 14*, pp. 255–293.

[78] Lenstra, Jr., H. W. and R. Tijdeman(eds.) (1982). *Computational Methods in Number Theory*. Number 154-155 in Mathematical Centre Tracts. Amsterdam, Holland: Mathematisch Centrum.

[79] Levy, S. (2001). *Crypto* (1st ed.). London, UK: Penguin.

[80] Leyland, P. (2003, April). Personal communication.

[81] Leyland, P., A. K. Lenstra, B. Dodson, A. Muffet, and S. Wagstaff (2002). MPQS with Three Large Primes. In *Proc. ANTS-V*, Volume 2369 of *LNCS*, pp. 446–460. Springer-Verlag.

[82] Li, S. and C. Pomerance (2000). Primitive Roots: A Survey. *Number Theoretic Methods – Future Trends* (to appear). Preprint available at `http://cm.bell-labs.com/cm/ms/who/carlp/PS/primitiverootstoo.ps`.

[83] Lim, C. H. and P. J. Lee (1997). A Key Recovery Attack on Discrete Log Based Schemes Using a Prime Order Subgroup. In *Proc. Crypto 1997*, Volume 1294 of *LNCS*, pp. 249–263. Springer-Verlag.

[84] Loudon, K. (1999). *Mastering Algorithms With C* (1st ed.). Sebastopol, CA: O'Reilly.

[85] Maurer, U. M. and S. Wolf (2000). The Diffie-Hellman Protocol. *Designs, Codes and Cryptography 19*, pp. 147–171.

[86] McCurley, K. S. (1990). The Discrete Logarithm Problem. *Proc. Symposium in Applied Mathematics 42*, pp. 49–74.

[87] Menezes, A., P. van Oorschot, and S. Vanstone (1996). *Handbook of Applied Cryptography* (1st ed.). Canada: CRC Press.

[88] Menezes, A. J., T. Okamoto, and S. A. Vanstone (1993). Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field. *IEEE Transactions on Information Theory IT-39*, pp. 1639–1646.

[89] Miller, V. S. (1986). Use of Elliptic Curves in Cryptography. In *Proc. Crypto 1985*, Volume 218 of *LNCS*, pp. 417–426. Springer-Verlag.

[90] Montgomery, P. L. (1995). A Block Lanczos Algorithm for Finding Dependencies over GF(2). In *Proc. EuroCrypt 1995*, Volume 921 of *LNCS*, pp. 106–120. Springer-Verlag.

[91] Morain, F. (1993). Analysing PMPQS – informal note. Available at `http://www.lix.polytechnique.fr/~morain/Articles/articles.english.html`.

[92] Morrison, M. A. and J. Brillhart (1975). A Method of Factoring and the Factorisation of F7. *Mathematics of Computation 29*, pp. 183–205.

[93] NIST (1993). FIPS PUB 46-2 Data Encryption Standard (DES). Available at `http://www.itl.nist.gov/fipspubsfip46-2.htm`.

[94] NIST (1994). FIPS PUB 186 Digital Signature Standard (DSS). Available at `http://www.itl.nist.gov/fipspubs/fip186.htm`.

[95] NIST (1998). SKIPJACK and KEA Algorithm Specifications. Available at `http://csrc.nist.gov/CryptoToolkit/skipjack/skipjack.pdf`.

[96] NIST (2001). FIPS PUB 197 Advanced Encryption Standard (AES). Available at `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[97] NIST (2002). Key Management Guideline. Second draft. Available at `http://csrc.nist.gov/encryption/kms/guideline-1.pdf`.

[98] Odlyzko, A. M. (1985). Discrete Logarithms in Finite Fields and their Cryptographic Significance. In *Proc. EuroCrypt 1984*, Volume 209 of *LNCS*, pp. 224–314. Springer-Verlag.

[99] Odlyzko, A. M. (1987). On the Complexity of Computing Discrete Logarithms and Factoring Integers. *Open Problems in Communication and Computation*, pp. 113–116.

[100] Odlyzko, A. M. (2000). Discrete Logarithms: The Past and the Future. *Designs, Codes, and Cryptography 19*, pp. 129–145.

[101] Peralta, R. (1985). Simultaneous Security of Bits in the Discrete Log. In *Proc. Eurocrypt 1985*, Volume 219 of *LNCS*, pp. 62–72. Springer-Verlag.

[102] Pfleeger, C. P. (1997). *Security In Computing* (2nd ed.). London, UK: Prentice Hall International.

[103] Pohlig, S. and M. Hellman (1978). An Improved Algorithm for Computing Logarithms over GF(p) and its Cryptographic Significance. *IEEE Transactions on Information Theory IT-24*, pp. 106–110.

[104] Pollard, J. M. (1978). Monte Carlo Methods for Index Computation (mod p). *Mathematics of Computation 32*, pp. 918–924.

[105] Pollard, J. M. (1993a). Factoring with Cubic Integers. In [73], pp. 4–10.

[106] Pollard, J. M. (1993b). The Lattice Sieve. In [73], pp. 43–49.

[107] Pomerance, C. (1983). Analysis and Comparison of some Integer Factoring Algorithms. In volume 1 of [78], pp. 89–139.

[108] Pomerance, C. and I. E. Shparlinski (2002). Smooth Orders and Cryptographic Applications. In *Proc. ANTS V*, Volume 2369 of *LNCS*, pp. 338–348. Springer-Verlag.

[109] Pomerance, C. and J. W. Smith (1992). Reduction of Huge, Sparse Linear Systems over Finite Fields via Created Catastrophes. *Experimental Mathematics 1*, pp. 89–94.

[110] Rivest, R., A. Shamir, and L. Adleman (1978). A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM 21*, pp. 120–126.

[111] Schindler, W., F. Koeune, and J.-J. Quisquater (2001). Improving Divide and Conquer Attacks Against Cryptosystems by Better Error Detection/Correction Strategies. In *Proc. 8th IMA Cryptography and Coding*, Volume 2260 of *LNCS*, pp. 245–267. Springer-Verlag.

[112] Schirokauer, O. (1999). Using Number Fields to Compute Logarithms in Finite Fields. *Mathematics of Computation 69*, pp. 1267–1283.

[113] Schirokauer, O., D. Weber, and T. Denny (1996). Discrete Logarithms: the Effectiveness of the Index Calculus Method. In *Proc. ANTS II*, Volume 1122 of *LNCS*, pp. 337–361. Springer-Verlag.

[114] Schneier, B. (1996). *Applied Cryptography* (2nd ed.). New York, NY: John Wiley & Sons, Inc.

[115] Semaev, I. A. (1998). An Algorithm for Evaluation of Discrete Logarithms in Some Nonprime Finite Fields. *Mathematics of Computation 67*, pp. 1679–1689.

[116] Seysen (1987). A Probabilistic Factorisation Algorithm with Quadratic Forms of Negative Discriminant. *Mathematics of Computation 48*, pp. 757–780.

[117] Shamir, A. and E. Tromer (2003). Factoring Large Numbers with the TWIRL Device. Available at `http://psifertex.com/download/twirl.pdf`.

[118] Shanks, D. (1971). Class Number, a Theory of Factorisation and Genera. In *Proc. Symposium on Pure Mathematics*, Volume 20, pp. 415–440. AMS.

[119] Shor, P. W. (1994). Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Proc. 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134. IEEE Computer Society Press.

[120] Shoup, V. (1997). Lower Bounds for Discrete Logarithms and Related Problems. In *Proc. EuroCrypt 1997*, Volume 1233 of *LNCS*, pp. 256–266. Springer-Verlag.

[121] Shoup, V. (2001). NTL - A Library for Number Theory (version 5.0c). Available at `http://www.shoup.net/ntl/`.

[122] Silverman, R. D. (2000). A Cost-Based Analysis of Symmetric and Asymmetric Key Lengths. Technical Report 13, RSA Security, Inc. Available at `http://rsasecurity.com/rsalabs/bulletins/bulletin13.html`.

[123] Smart, N. (2002). *Cryptography - An Introduction* (1st ed.). Maidenhead, UK: McGraw Hill.

[124] Stephens, N. M. (1986). Lenstra's Factorisation Method Based on Elliptic Curves. In *Proc. Crypto 1985*, Volume 2182 of *LNCS*, pp. 409–416. Springer-Verlag.

[125] Teske, E. (1998). Speeding up Pollard's Method for Computing Discrete Logarithms. In *Proc. ANTS III*, Volume 1423 of *LNCS*, pp. 541–553. Springer-Verlag.

[126] Teske, E. (2001). Square-Root Algorithms for the Discrete Logarithm Problem A Survey. In *Proc. International Conference Organised by the Stefan Banach International Mathematical Center*, Warsaw, Poland, pp. 283–301. Walter de Gruyter.

[127] Thomé, E. (2001). Computation of Discrete Logarithms in GF(2^{607}). In *Proc. AsiaCrypt 2001*, Volume 2248 of *LNCS*, pp. 107–124. Springer-Verlag.

[128] van Oorschot, P. (1990). A Comparison of Practical Public Key Cryptosystems Based on Integer Factorisation and Discrete Logarithms. In *Proc. Crypto 1990*, Volume 537 of *LNCS*, pp. 576–581. Springer-Verlag.

[129] van Oorschot, P. and M. J. Wiener (1996). On Diffie-Hellman Key Agreement with Short Exponents. In *Proc. EuroCrypt 1996*, Volume 1070 of *LNCS*, pp. 332–343. Springer-Verlag.

[130] Wagstaff, Jr., S. S. (2002). *Cryptanalysis of Number Theoretic Ciphers* (1st ed.). London, UK: CRC Press.

[131] Weber, D. (1995). An Implementation of the General Number Field Sieve to Compute Discrete Logarithms Mod p. In *Proc. EuroCrypt 1995*, Volume 921 of *LNCS*, pp. 95–105. Springer-Verlag.

[132] Weber, D. (1996). Computing Discrete Logarithms with the General Number Field Sieve. In *Proc. ANTS II*, Volume 1122 of *LNCS*, pp. 391–403. Springer-Verlag.

[133] Weber, D. (1998). Computing Discrete Logarithms with Quadratic Number Rings. In *Proc. EuroCrypt 1998*, Volume 1403 of *LNCS*, pp. 171–183. Springer-Verlag.

[134] Weber, D. (2003, February). Personal communication.

[135] Wells, Jr., A. L. (1984). A Polynomial Form for Logarithms Modulo a Prime. *IEEE Transactions on Information Theory 30*, pp. 845–846.

[136] Western, A. E. and J. C. P. Miller (1968). *Tables of Indices and Primitive Roots.* Volume 9 of *Royal Society Mathematical Tables.* Cambridge, UK: Cambridge University Press.

[137] Wiedemann, D. H. (1986). Solving Sparse Linear Equations Over Finite Fields. *IEEE Transactions on Information Theory 32*, pp. 54–62.

[138] Zierler, N. (1974). A Conversion Algorithm for Logarithms on GF(2^n). *Journal of Pure and Applied Algebra 4*, pp. 353–356.